

Real-Time Systems and Programming Languages

INTERNATIONAL COMPUTER SCIENCE SERIES

Consulting Editor **A D McGettrick** University of Strathclyde

SELECTED TITLES IN THE SERIES

- Concurrent Systems: An Integrated Approach to Operating Systems, Database,
and Distributed Systems (2nd edn) *J Bacon*
- Programming Language Essentials *H E Bal and D Grune*
- Programming in Ada 95 (2nd edn) *J G P Barnes*
- Java Gently (3rd edn) *J Bishop*
- Software Design *D Budgen*
- Concurrent Programming *A Burns and G Davies*
- Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and
Real-Time POSIX (3rd edn) *A Burns and A J Wellings*
- Comparative Programming Languages, (3rd edn) *Wilson and Clark, updated by Clark*
- Distributed Systems: Concepts and Design (3rd edn) *G Coulouris, J Dollimore
and T Kindberg*
- Principles of Object-Oriented Software Development (2nd edn) *A Eliëns*
- Fortran 90 Programming *T M R Ellis, I R Philips and T M Lahey*
- Program Verification *N Francez*
- Introduction to Programming using SML *M Hansen and H Rischel*
- Functional C *P Hartel and H Muller*
- Ada 95 for C and C++ Programmers *S Johnston*
- Algorithms and Data Structures: Design, Correctness, Analysis (2nd edn) *J Kingston*
- Introductory Logic and Sets for Computer Scientists *N Nissanke*
- Human-Computer Interaction *J Preece et al.*
- Algorithms: a Functional Programming Approach *F Rabhi and G Lapalme*
- Ada 95 From the Beginning (3rd edn) *J Skansholm*
- C++ From the Beginning *J Skansholm*
- Java From the Beginning *J Skansholm*
- Software Engineering (6th edn) *I Sommerville*
- Object-Oriented Programming in Eiffel (2nd edn) *P Thomas and R Weedon*
- Miranda: The Craft of Functional Programming *S Thompson*
- Haskell: The Craft of Functional Programming (2nd edn) *S Thompson*
- Discrete Mathematics for Computer Scientists (2nd edn) *J K Truss*
- Compiler Design *R Wilhelm and D Maurer*
- Discover Delphi: Programming Principles Explained *S Williams and S Walmsley*
- Software Engineering with B *J B Wordsworth*

Real-Time Systems and Programming Languages

Ada, Real-Time Java and C/Real-Time POSIX

Fourth Edition

Alan Burns and Andy Wellings

University of York



ADDISON-WESLEY

An imprint of Pearson Education

Harlow, England • London • New York • Boston • San Francisco • Toronto • Sydney • Singapore • Hong Kong
Tokyo • Seoul • Taipei • New Delhi • Cape Town • Madrid • Mexico City • Amsterdam • Munich • Paris • Milan

Pearson Education Limited

Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsoned.co.uk

First published 1989
Second edition 1997
Third edition 2001

Fourth edition published 2009

© Pearson Education Limited 1989, 2009

The rights of Alan Burns and Andy Wellings to be identified as authors of this work have been asserted by them in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without either the prior written permission of the publisher or a licence permitting restricted copying in the United Kingdom issued by the Copyright Licensing Agency Ltd, Saffron House, 6–10 Kirby Street, London EC1N 8TS.

All trademarks used herein are the property of their respective owners. The use of any trademark in this text does not vest in the author or publisher any trademark ownership rights in such trademarks, nor does the use of such trademarks imply any affiliation with or endorsement of this book by such owners.

ISBN: 978-0-321-41745-9

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library

Library of Congress Cataloging-in-Publication Data

A catalogue record for this book can be obtained from the Library of Congress

10 9 8 7 6 5 4 3 2 1
13 12 11 10 09

Typeset in Times Roman 10/12 by 73
Printed in Great Britain by Henry Ling Ltd., at the Dorset Press, Dorchester, Dorset

The publisher's policy is to use paper manufactured from sustainable forests.

Contents

Preface	xiii
1 Introduction to real-time systems	1
1.1 Definition of a real-time system	2
1.2 Examples of real-time systems	4
1.3 Characteristics of real-time systems	9
1.4 Development cycle for real-time systems	15
1.5 Languages for programming real-time systems	20
Summary	23
Further reading	25
Exercises	26
2 Reliability and fault tolerance	27
2.1 Reliability, failure and faults	28
2.2 Failure modes	31
2.3 Fault prevention and fault tolerance	33
2.4 <i>N</i> -version programming	36
2.5 Software dynamic redundancy	41
2.6 The recovery block approach to software fault tolerance	46
2.7 A comparison between <i>N</i> -version programming and recovery blocks	49
2.8 Dynamic redundancy and exceptions	50
2.9 Measuring and predicting the reliability of software	52
2.10 Safety, reliability and dependability	53
Summary	55
Further reading	56
Exercises	57
3 Exceptions and exception handling	59
3.1 Exception handling in older real-time languages	60
3.2 Modern exception handling	62
3.3 Exception handling in Ada, Java and C	69
3.4 Recovery blocks and exceptions	85

Summary	88
Further reading	89
Exercises	89
4 Concurrent programming	95
4.1 Processes and tasks/threads	96
4.2 Concurrent execution	99
4.3 Task representation	103
4.4 Concurrent execution in Ada	105
4.5 Concurrent execution in Java	111
4.6 Concurrent execution in C/Real-Time POSIX	116
4.7 Multiprocessor and distributed systems	121
4.8 A simple embedded system	125
4.9 Language-supported versus operating-system-supported concurrency	131
Summary	132
Further reading	133
Exercises	133
5 Shared variable-based synchronization and communication	137
5.1 Mutual exclusion and condition synchronization	138
5.2 Busy waiting	139
5.3 Suspend and resume	142
5.4 Semaphores	145
5.5 Conditional critical regions	156
5.6 Monitors	157
5.7 Mutexes and condition variables in C/Real-Time POSIX	160
5.8 Protected objects in Ada	163
5.9 Synchronized methods in Java	171
5.10 Shared memory multiprocessors	179
5.11 Simple embedded system revisited	183
Summary	185
Further reading	186
Exercises	187
6 Message-based synchronization and communication	193
6.1 Process synchronization	193
6.2 Task naming and message structure	195
6.3 Message passing in Ada	196
6.4 Selective waiting	201
6.5 The Ada select statement	202
6.6 Non-determinism, selective waiting and synchronization primitives	205
6.7 C/Real-Time POSIX message queues	206
6.8 Distributed systems	210
6.9 Simple embedded system revisited	219
Summary	220
Further reading	221
Exercises	222

7	Atomic actions, concurrent tasks and reliability	227
7.1	Atomic actions	228
7.2	Atomic actions in C/Real-Time POSIX, Ada and Real-Time Java ...	232
7.3	Recoverable atomic actions	240
7.4	Asynchronous notification	245
7.5	Asynchronous notification in C/Real-Time POSIX	247
7.6	Asynchronous notification in Ada	255
7.7	Asynchronous notification in Real-Time Java	266
	Summary	278
	Further reading	280
	Exercises	280
8	Resource control	285
8.1	Resource control and atomic actions	286
8.2	Resource management	286
8.3	Expressive power and ease of use	287
8.4	The requeue facility	296
8.5	Asymmetric naming and security	302
8.6	Resource usage	303
8.7	Deadlock	304
	Summary	304
	Further reading	305
	Exercises	305
9	Real-time facilities	307
9.1	The notion of time	307
9.2	Access to a clock	309
9.3	Delaying a task	317
9.4	Programming timeouts	320
9.5	Specifying timing requirements	326
9.6	Temporal scopes	328
	Summary	332
	Further reading	333
	Exercises	333
10	Programming real-time abstractions	335
10.1	Real-time tasks	336
10.2	Programming periodic activities	338
10.3	Programming aperiodic and sporadic activities	341
10.4	The role of real-time events and their handlers	344
10.5	Controlling input and output jitter	349
10.6	Other approaches for supporting temporal scopes	356
	Summary	363
	Further reading	364
	Exercises	364

11 Scheduling real-time systems	365
11.1 The cyclic executive approach	366
11.2 Task-based scheduling	367
11.3 Fixed-priority scheduling (FPS)	370
11.4 Utilization-based schedulability tests for FPS	371
11.5 Response time analysis (RTA) for FPS	374
11.6 Sporadic and aperiodic tasks	378
11.7 Task systems with $D < T$	380
11.8 Task interactions and blocking	382
11.9 Priority ceiling protocols	386
11.10 An extendible task model for FPS	390
11.11 Earliest deadline first (EDF) scheduling	400
11.12 Dynamic systems and online analysis	405
11.13 Worst-case execution time	407
11.14 Multiprocessor scheduling	408
11.15 Scheduling for power-aware systems	413
11.16 Incorporating system overheads	414
Summary	419
Further reading	420
Exercises	421
 12 Programming schedulable systems	 425
12.1 Programming cyclic executives	425
12.2 Programming preemptive priority-based systems	426
12.3 Ada and fixed-priority scheduling	427
12.4 The Ada Ravenscar profile	430
12.5 Dynamic priorities and other Ada facilities	434
12.6 C/Real-Time POSIX and fixed-priority scheduling	436
12.7 Real-Time Java and fixed-priority scheduling	438
12.8 Programming EDF systems	443
12.9 Mixed scheduling	453
Summary	454
Further reading	454
Exercises	455
 13 Tolerating timing faults	 457
13.1 Dynamic redundancy and timing faults	457
13.2 Deadline miss detection	459
13.3 Overrun of worst-case execution time	467
13.4 Overrun of sporadic events	471
13.5 Overrun of resource usage	474
13.6 Damage confinement	475
13.7 Error recovery	485
Summary	492
Further reading	493
Exercises	493

14 Low-level programming	495
14.1 Hardware input/output mechanisms	495
14.2 Language requirements	502
14.3 Ada	504
14.4 Real-Time Java	514
14.5 C and older real-time languages	517
14.6 Scheduling device drivers	519
14.7 Memory management	521
Summary	527
Further reading	528
Exercises	528
15 Mine control case study	533
15.1 Mine drainage	533
15.2 The HRT-HOOD design method	538
15.3 The logical architecture design	539
15.4 The physical architecture design	545
15.5 Translation to Ada	546
15.6 Translation to Real-Time Java	564
15.7 Fault tolerance and distribution	570
Summary	572
Further reading	572
Exercises	573
16 Conclusions	575
References	579
Index	587

Supporting resources

Visit www.pearsoned.co.uk/burns to find valuable online resources

For instructors

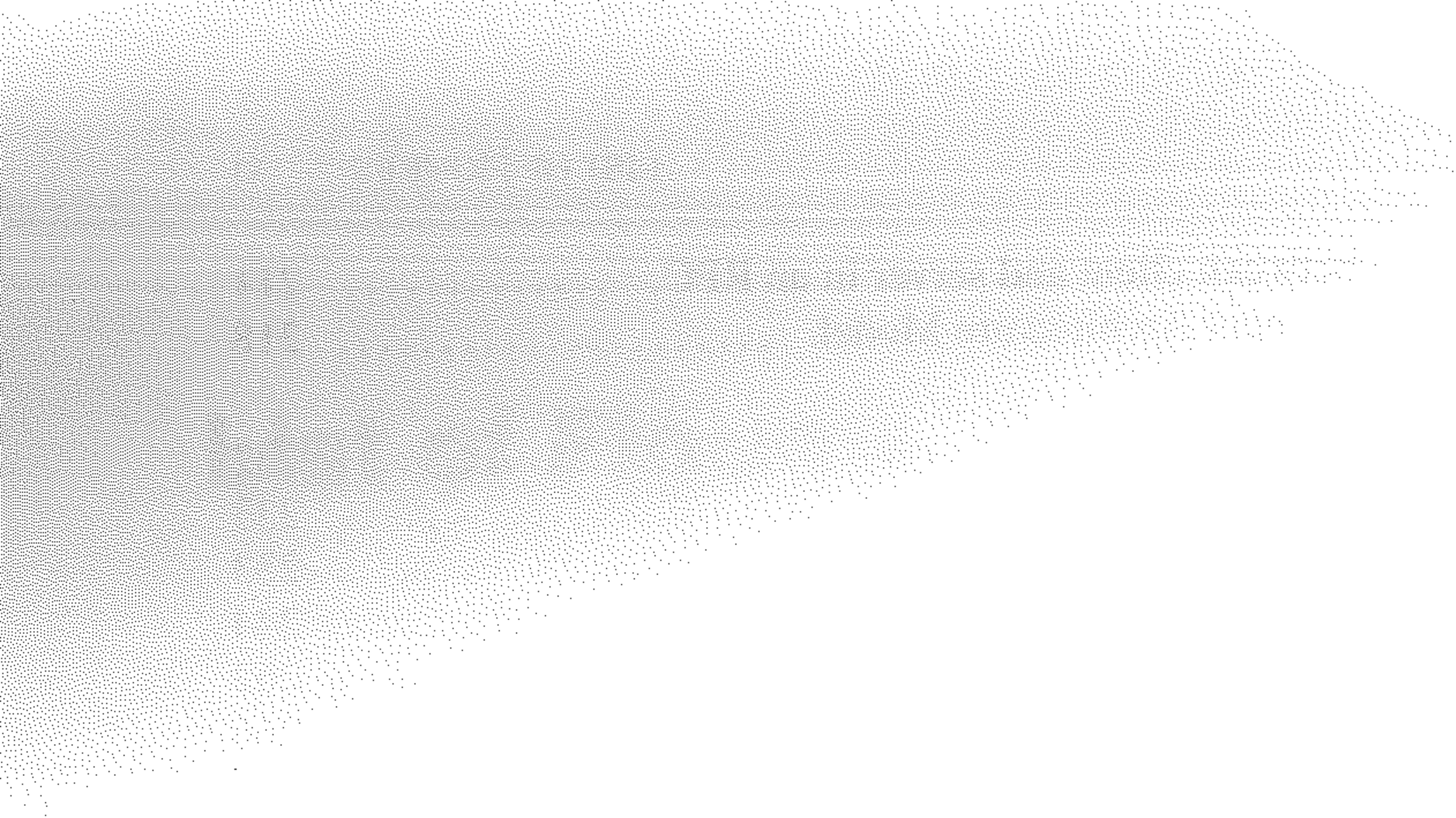
- Solutions to exercises
- Example examination questions
- Code fragments
- PowerPoint slides

For more information please contact your local Pearson Education sales representative or visit www.pearsoned.co.uk/burns

List of Figures

1.1	A fluid control system.	4
1.2	A process control system.	5
1.3	A production control system.	6
1.4	A command and control system.	7
1.5	A typical embedded system.	8
1.6	A simple controller.	12
1.7	A simple computerized controller.	13
1.8	Aspects of real-time systems.	24
2.1	Fault, error, failure, fault chain.	29
2.2	Failure mode classification.	32
2.3	Graceful degradation and recovery in an air traffic control system.	35
2.4	N-version programming.	38
2.5	Consistent comparison problem with three versions.	40
2.6	The domino effect.	45
2.7	Recovery block mechanism.	47
2.8	An ideal fault-tolerant component.	52
2.9	Aspects of dependability.	54
2.10	Dependability terminology.	55
2.11	Concurrent execution of four processes for Exercise 2.5.	57
3.1	The resumption model.	67
3.2	The termination model.	69
3.3	The Java predefined Throwable class hierarchy.	78
4.1	Simple state diagram for a task.	98
4.2	State diagram for a task.	101
4.3	Fork and join.	104
4.4	Cobegin.	105
4.5	A simple embedded system.	126
5.1	State diagram for a task.	145

6.1	The relationship between client and server in an RPC.	211
6.2	The Object Management Architecture Model.	218
7.1	Nested atomic actions.	232
7.2	The structure of an action controller.	233
7.3	Using the action controller.	233
7.4	An exception in a nested atomic action.	244
7.5	Simple state transition diagram for a conversation.	262
7.6	Simple state transition diagram illustrating forward error recovery.	266
8.1	A network router.	300
9.1	Delay times.	318
9.2	Temporal scopes.	328
10.1	A simple task with input and output jitter constraints.	350
10.2	Three threads implementing input and output jitter constraints.	352
10.3	The Logical Execution Time model.	362
11.1	Time-line for task set.	366
11.2	Time-line for task set A.	372
11.3	Gantt chart for task set A.	372
11.4	Time-line for task set C.	373
11.5	Gantt chart for task set D.	378
11.6	Example of priority inversion.	383
11.7	Example of priority inheritance.	385
11.8	Example of priority inheritance – OCPP.	388
11.9	Example of priority inheritance – ICPP.	389
11.10	Releases of sporadic tasks.	391
11.11	PDC example.	402
11.12	Overheads when executing tasks.	415
11.13	The behaviour of four periodic tasks in Exercise 11.4.	422
14.1	Architecture with separate buses.	496
14.2	Memory-mapped architecture.	496
15.1	A mine drainage control system.	534
15.2	Graph showing external devices.	535
15.3	First-level component decomposition of the control system.	540
15.4	PumpController related interfaces.	540
15.5	Other defined interfaces.	541
15.6	Hierarchical decomposition of the PumpController object.	542
15.7	Decomposition of the HighLowWaterSensors.	543
15.8	Hierarchical decomposition of the EnvironmentMonitor.	544
15.9	State transition diagram for the motor.	552



Preface

In 1981, a software error caused a stationary robot to move suddenly and with impressive speed to the edge of its operational area. A nearby worker was crushed to death.

This is just one example of the hazards of embedded real-time systems. It is unfortunately not an isolated incident. Every month the newsletter *Software Engineering Notes* has pages of examples of events in which the malfunctioning of real-time systems has put the public or the environment at risk. What these sobering descriptions illustrate is that there is a need to take a system-wide view of embedded systems. Indeed it can be argued that there is the requirement for real-time systems to be recognized as a distinct engineering discipline. This book is a contribution towards the development of this discipline. It cannot, of course, cover all the topics that are apposite to the study of real-time systems engineering; it does, however, present a comprehensive description and assessment of the programming languages and operating system standards used in this domain. Particular emphasis is placed on language primitives and their role in the production of reliable, safe and dependable software.

Audience

The book is aimed at Final Year and Masters students in Computer Science and related disciplines. It has also been written with the professional software engineer, and real-time systems engineer, in mind. Readers are assumed to have knowledge of sequential programming languages and some prior experience of C, Java and Ada, and to be familiar with the basic tenets of software engineering. The material presented reflects the content of courses developed over a number of years by the authors at various universities and for industry. These courses specifically address real-time systems and programming languages.

Structure and content

In order to give the chapters continuity, three programming languages are considered in detail: Ada, Java and C. These languages have been chosen because they are actually used

for software production. As C is a sequential language, it is used in conjunction with the POSIX family of operating system interfaces (in particular, the real-time extensions). To emphasize this, it will be referred to as *C/Real-Time POSIX*. As Java was not originally intended to be used for real-time systems development it must be augmented with the facilities of the Real-Time Specification for Java (RTSJ). To emphasize this, it will be referred to as *Real-Time Java*. Ada was designed for real-time systems development. Other theoretical or experimental languages are discussed when they offer primitives not available within the core languages. Practitioners who are primarily interested in only one of these languages should find sufficient material for their needs. The authors believe that a full appreciation of a language like Ada or Java, say, can only be obtained through a comparative study of their facilities.

In all, the book contains 16 chapters, the first 8 of which are loosely organized into the following three groups. Chapter 1 represents an extended introduction. The characteristics and requirements of real-time systems are presented, then an overview of the design of such systems is given. Design is not the primary focus of this book; nevertheless, it is important to discuss implementation within an appropriate context.

Chapters 2 and 3 concern themselves with the production of reliable software components. Although consideration is given to fault prevention, attention is primarily focused on fault tolerance. Both forward and backward error recovery techniques are considered. The use of an exception-handling facility is discussed in Chapter 3. Both resumption and termination models are described, as are the language primitives found in Ada and Java.

Real-time systems are inherently concurrent, and therefore the study of this aspect of programming languages is fundamental. Chapter 4 introduces the notions of process, tasks and thread and reviews the many different models that are used by language and operating system designers. The term *task* is used generically to represent a concurrent activity. Communication between tasks is considered in the following two chapters. Firstly shared-variable methods are described including the use of semaphores, monitors, mutexes and protected objects. Message-based models are also important in modern languages; combining as they do communication and synchronization. These models are covered in Chapter 6. Particular attention is given to the rendezvous primitives of Ada.

It is debatable whether issues of reliability or concurrency should have been considered first within the book. Both authors have experimented with reversing the order and have found little to choose between the two possible approaches. The book can in fact be used in either mode with only one or two topics being 'out of place'. The decision to cover reliability first reflects the authors' belief that safety is the predominant requirement of real-time systems.

The next grouping incorporates Chapters 7 and 8. In general, the relationship between system tasks can be described as either cooperating (to achieve a common goal) or competing (to acquire a shared resource). Chapter 7 extends the earlier discussions on fault tolerance by describing how reliable task cooperation can be programmed. Central to this discussion is the notion of an *atomic action* and asynchronous notification techniques. Competing processes are considered in the following chapter. An assessment is given of different language features. One important topic here is the distinction between conditional and avoidance synchronization within the concurrency model.

Temporal requirements constitute the distinguishing characteristic of real-time systems. It is therefore appropriate that a large portion of the book focuses on how to

meet them. Chapter 9 introduces the notions of time and clocks, along with the role of *temporal scopes* for specifying timing constraints. Chapter 10 then shows the common programming abstractions that are used to represent these constraints. Ensuring that timing constraints can be met at run-time requires real-time scheduling. Hard real-time systems have timing constraints that must be satisfied; soft systems can occasionally fail to perform adequately. The mathematical analysis that underpins real-time scheduling is covered in Chapter 11: both priority and deadline-based scheduling is considered. The support provided for programming schedulable system is then covered in Chapter 12. Chapter 13 brings together fault tolerance and real-time, focusing on the options available to the programmer when timing faults occur at run-time.

One important requirement of many real-time systems is that they incorporate external devices that must be programmed (that is, controlled) as part of the application software. This low-level programming is at variance with the abstract approach to software production that characterizes software engineering. Chapter 14 considers ways in which low-level facilities can be successfully incorporated into high-level languages.

The final major chapter of the book is a case study. An example from a mine control system is used. Inevitably a single scaled down study cannot illustrate all the issues covered in the previous chapters; in particular factors such as size and complexity are not addressed. Nevertheless, the case study does cover many important aspects of real-time systems.

All chapters have summaries and further reading lists. Most also have lists of exercises. These have been chosen to help readers consolidate their understanding of the material presented in each chapter. They mostly represent exercises that have been used by the authors for assessment purposes.

Ada, Java and C

The Ada examples in this book conform to the Ada 2005 ISO/ANSI standard. The Java examples conform to the Java 5 platform along with the Real-Time Specification for Java extensions (Version 1.0.2). The C examples conform to ANSI C, and the POSIX primitives are those given in the IEEE Std 1003.1, 2004 Edition.

To facilitate easy identification of the three languages, different presentation styles are used. Ada is presented with keywords in bold lower case; program identifiers are given in mixed case. C has keywords unbolded and identifiers in lower case. To distinguish Java from C, Java keywords are bolded and identifiers are mixed case.

Changes from the Third Edition

Over the last 20 years, real-time technology has advanced considerably. Consequently this book has been creeping up in size. The fourth edition has added significant new material, and, as a result, we have restructured the book to remove some of the material that can be found elsewhere.

- The material on design has been pruned and incorporated into the Introduction. The advent of the UML real-time profile meant that we could no longer give this

topic the attention it deserved. As design issues are not our focus, we decided it was best to cover less.

- The material on programming in the small and large, which provided the introduction to sequential programming in C, Java and Ada has been removed completely. We felt that the material is best served by specialist books on programming in these languages.
- We have removed discussions of occam2 and Modula to an appendix that can be obtained from the book's web page. Although these languages are no longer in widespread use, we believe they are historically important.
- The chapter on Distributed Systems has been removed, and the main topics have been distributed throughout other chapters in the book. Again, we felt that we were unable to do justice to this topic, but did not want to lose some of the important real-time issues.
- The removal of occam2 from the book left the Execution Environment chapter weak, so again we have removed it and distributed the remaining material throughout the book, mainly to Chapter 14.
- The main new material has been introduced into the part of the book that focuses on timing issues. What was previously two chapters has now been expanded into five chapters.

We have also update throughout our treatment of Ada, Real-Time Java and C/Real-Time POSIX to reflect the recent revisions to the associated definitions and standards.

Teaching Aids

This text is supported by further material available via the following WWW site:

<http://www.pearsoned.co.uk/burns>

Overhead projection foil layouts are available for many parts of the book. Also available are solutions to some of the exercises. We will, over time, add further exercises, and where appropriate new examples and additional teaching material. Teachers/lecturers who make use of this book are invited to contribute to these web pages.

Real-Time Systems Research at York

Alan Burns and Andy Wellings are members of the Real-Time Systems Research Group in the Department of Computer Science at the University of York (UK). This group undertakes research into all aspects of the design, implementation and analysis of real-time systems.

Specifically, the group is addressing: formal and structured methods for development, scheduling theories, reuse, language design, kernel design, communication protocols, distributed and parallel architectures, and program code analysis. The aim of the

group is to undertake fundamental research, and to bring into engineering practice modern techniques, methods and tools. Areas of application of our work include space and avionic systems, engine controllers, vehicle control and multi-media systems. Further information about the group's activities can be found via:

<http://www.cs.york.ac.uk/rts>

Acknowledgements for the First Edition

The material in this book has been developed over the last five years and presented to many third year and MSc students at the Universities of Bradford and York, taking Computer Science or Electronics degrees. We would like to acknowledge their contribution to the end product, for without them this book would never have been written.

Many people have read and commented on a first draft of the book. In particular we would like to thank: Martin Atkins, Chris Hoggarth, Andy Hutcheon, Andrew Lister and Jim Welsh. We would also like to thank our colleagues at our respective Universities for providing us with a stimulating environment and for many enlightening discussions, particularly Ljerka Beus-Dukic, Geoff Davies, John McDermid, Gary Morgan, Rick Pack, Rob Stone and Hussein Zedan.

During 1988 Alan Burns was on sabbatical at the Universities of Queensland and Houston. We would like to thank all staff at these institutions particularly Andrew Lister, Charles McKay and Pat Rogers.

This book would not have been possible without the use of electronic mail over JANET. We would like to thank the Computer Board of the United Kingdom University Grants Council and the Science and Engineering Research Council for providing this invaluable service.

Finally we would like to give special thanks to Sylvia Holmes and Carol Burns. Sylvia for the many hours she has spent painstakingly proof reading the final manuscript, and Carol for the many evenings she has tolerated our meetings and discussions.

Acknowledgements for the Second Edition

Many people have helped in the production of the Second Edition of this book. In particular we would like to thank: Alejandro Alonso, Angel Alvarez, Sergio Arevalo, Neil Audsley, Martin Dorey, Michael González Harbour, Stuart Mitchell, Gary Morgan, Offer Pazy and Juan de la Puente.

Acknowledgements for the Third Edition

We would like to thank the Real-Time Java Expert Group for the open manner in which they have developed the Real-Time Java Specification. Thanks are also due to Angel Alvarez, Jose Alvarez, Neil Audsley, Iain Bate, Jorge Diaz-Herrera, David Duke, Alan Grigg, Ian Hayes, George Lima, Greg Murphy, Peter Puschner and Pat Rogers who all provided us with help, of one form or another, during the writing of this edition.

We also wish to acknowledge the very helpful comments given by the technical reviewers, Jorge Díaz-Herrera, Jörgen Hansson and Robert Holton, on the first draft of this edition.

Finally, we would like to thank all those people who gave us comments on the Second Edition of the book.

Acknowledgements for the Fourth Edition

We would like to thank all those people who gave us comments on the Third Edition of the book, particularly Yolande Berbers. We also wish to thank the other members of the Ada Rapporteur Group, the Technical Interpretation Committee for the Real-Time Specification for Java (particularly Peter Dibble), and the Java Expert Groups (JSR 282 and 301) for their help in understanding the nuances of Ada and Real-Time Java. Sanjoy Baruah, Michael González Harbour and Bev Littlewood have also given us invaluable help with multiprocessor scheduling, Real-Time POSIX and software reliability estimations, respectively.

We would like to acknowledge the past and present members of the Real-Time Systems Group at York for their continuing contribution to the material presented.

Finally, a special thanks go to all the students who have taken our Real-Time Systems module as part of their degree course at York. Their continuous comments and criticisms have helped keep us on our toes!

Alan Burns & Andy Wellings
April 2009

Chapter 1

Introduction to real-time systems

1.1	Definition of a real-time system	1.4	Development cycle for real-time systems
1.2	Examples of real-time systems	1.5	Languages for programming real-time systems
1.3	Characteristics of real-time systems		Summary
			Further reading
			Exercises

As computers become smaller, faster, more reliable and cheaper, so their range of application widens. Built initially as equation solvers, their influence has extended into all walks of life, from washing machines to air traffic control. One of the fastest expanding areas of computer exploitation is that involving applications whose prime function is *not* that of information processing, but which nevertheless require information processing in order to carry out their prime function. A microprocessor-controlled washing machine is a good example of such a system. Here, the prime function is to wash clothes; however, depending on the type of clothes to be washed, different ‘wash programs’ must be executed. These types of computer applications are generically called **real-time** or **embedded**. It has been estimated that 99% of the worldwide production of microprocessors is used in embedded systems. These embedded systems place particular requirements on the computer languages needed to program them, as they have different characteristics from the more traditional information processing systems.

This book is concerned with embedded computer systems and their programming languages. It studies the particular characteristics of these systems and discusses how modern real-time programming languages and operating systems have evolved. In order to give the chapters continuity, three programming languages are considered in detail: Ada, Java and C. These languages have been chosen because they are actually used for software production. As C is a sequential language, it is used in conjunction with the POSIX family of operating system interfaces (in particular, the real-time extensions). To emphasize this, it will be referred to a *C/Real-Time POSIX*. As Java was not originally intended to be used for real-time systems development it must be augmented with the facilities of the Real-Time Specification for Java (RTSJ). To emphasize

this, it will be referred to as *Real-Time Java*. Ada was designed for systems development; however, some of its libraries (for example, the `Real_Time` package) are optional. Here we assume these libraries are supported.

1.1 Definition of a real-time system

Before proceeding further, it is worth trying to define the phrase ‘real-time system’ more precisely. There are many interpretations of the exact nature of a real-time system; however, they all have in common the notion of response time – the time taken for the system to generate output from some associated input. The *Oxford Dictionary of Computing* gives the following definition of a real-time system:

Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.

Here, the word ‘timeliness’ is taken in the context of the total system. For example, in a missile guidance system, output is required within a few milliseconds, whereas in a computer-controlled car assembly line, the response may be required only within a second. To illustrate the various ways in which ‘real-time’ systems are defined, two further definitions will be given. Young (1982) defines a real-time system to be:

any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period.

Another definition is (Randell et al., 1995):

A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment.

In their most general sense, all these definitions cover a very wide range of computer activities. For example, an operating system like Windows may be considered real-time in that when a user enters a command he or she will expect a response within a few seconds. Fortunately, it is usually not a disaster if the response is not forthcoming. These types of system can be distinguished from those where *failure* to respond can be considered just as bad as a wrong response. Indeed, for some, it is this aspect that distinguishes a real-time system from others where response time is important but not crucial. Consequently, *the correctness of a real-time system depends not only on the logical result of the computation, but also on the time at which the results are produced*. Practitioners in the field of real-time computer system design often distinguish between **hard** and **soft** real-time systems. Hard real-time systems are those where it is absolutely imperative that responses occur within the specified deadline. Soft real-time systems are those where response times are important but the system will still function correctly if deadlines are occasionally missed. Soft systems can themselves be distinguished from

interactive ones in which there are no explicit deadlines. For example, the flight control system of a combat aircraft is a hard real-time system because a missed deadline could lead to a catastrophe, whereas a data acquisition system for a process control application is soft, as it may be defined to sample an input sensor at regular intervals but to tolerate intermittent delays. Of course, many systems will have both hard and soft real-time sub-systems. Indeed, some services may have both a soft and a hard deadline. For example, a response to some warning event may have a soft deadline of 50 ms (for an optimally efficient reaction) and a hard deadline of 200 ms (to guarantee that no damage to equipment or personnel takes place). Between 50 ms and 200 ms, the 'value' (or utility) of the output decreases.

As these definitions and examples illustrate, the use of the term 'soft' does not imply a single type of requirement, but incorporates a number of different properties. For example:

- the deadline can be missed occasionally (typically with an upper limit of misses within a defined interval);
- the service can occasionally be delivered late (again, with an upper limit on tardiness).

A deadline that can be missed occasionally, but in which there is no benefit from late delivery, is called **firm**. In some real-time systems, optional firm components may be given probabilistic requirements (for example, a hard service must produce an output every 300 ms; at least 80% of the time this output will be produced by a firm component, X; on other occasions, a hard, but functionally much simpler component, Y, will be used).

In this book, the term 'real-time system' is used to mean both soft and hard real-time. Where discussion is concerned specifically with hard real-time systems, the term 'hard real-time' will be used explicitly.

In a hard or soft real-time system, the computer is usually interfaced directly to some physical equipment and is dedicated to monitoring or controlling the operation of that equipment. A key feature of all these applications is the role of the computer as an information processing component within a larger engineering system. It is for this reason that such applications have become known as **embedded computer systems**.

Another means of classifying the role that time has in real-time systems is to distinguish between **reactive systems** and **time-aware systems**. Time-aware systems make explicit references to the time frame of the enclosing environment. For example, if a bank safe's doors are to be locked from midnight to nine o'clock in the morning then these absolute time values must be available to the system. By comparison, a reactive system is typically concerned with relative times: an output has to be produced within 50 ms of an associated input. The key requirement of a reactive system is that it 'keeps up with the environment'. Often reactive systems are also control systems and hence they need to be synchronized with their environment. In particular, input sampling and output signalling must be done very regularly with controlled variability – there is a need to bound what is called **input jitter** and **output jitter**.

In order for a reactive systems to 'keep up with its environment' they are often structured to be **time-triggered**. All computation activities are **periodic** in that they have a defined cycle time, for example 50 ms, and are released for execution by an internal clock. The alternative to time-triggered is **event-triggered** in which the environment

explicitly controls (perhaps via an interrupt) the release for execution of some software activity. These activities are termed **aperiodic** or, if there is a bound on how often the releasing event can occur in any time interval, **sporadic**. Many systems will contain periodic and sporadic activities. However, some design approaches restrict the software architecture so that there are only time-triggered activities; ‘events’ must be polled for (that is, examined via a periodic activity).

Again, in this book a broad definition of ‘real-time system’ is assumed. The term is taken to include reactive and time-aware systems that may have both time-triggered and event-triggered invocations of work. Periodic, aperiodic and sporadic activities are all likely to be present in the same system.

1.2 Examples of real-time systems

Having defined what is meant by a real-time systems, some examples of their use are now given.

1.2.1 Process control

The first use of a computer as a component in a larger engineering system occurred in the process control industry in the early 1960s. Nowadays, the use of microprocessors is the norm. Consider the simple example shown in Figure 1.1, where the computer performs a single activity: that of ensuring an even flow of liquid in a pipe by controlling a valve. On detecting an increase in flow, the computer must respond by altering the valve angle; this response must occur within a finite period if the equipment at the receiving end of

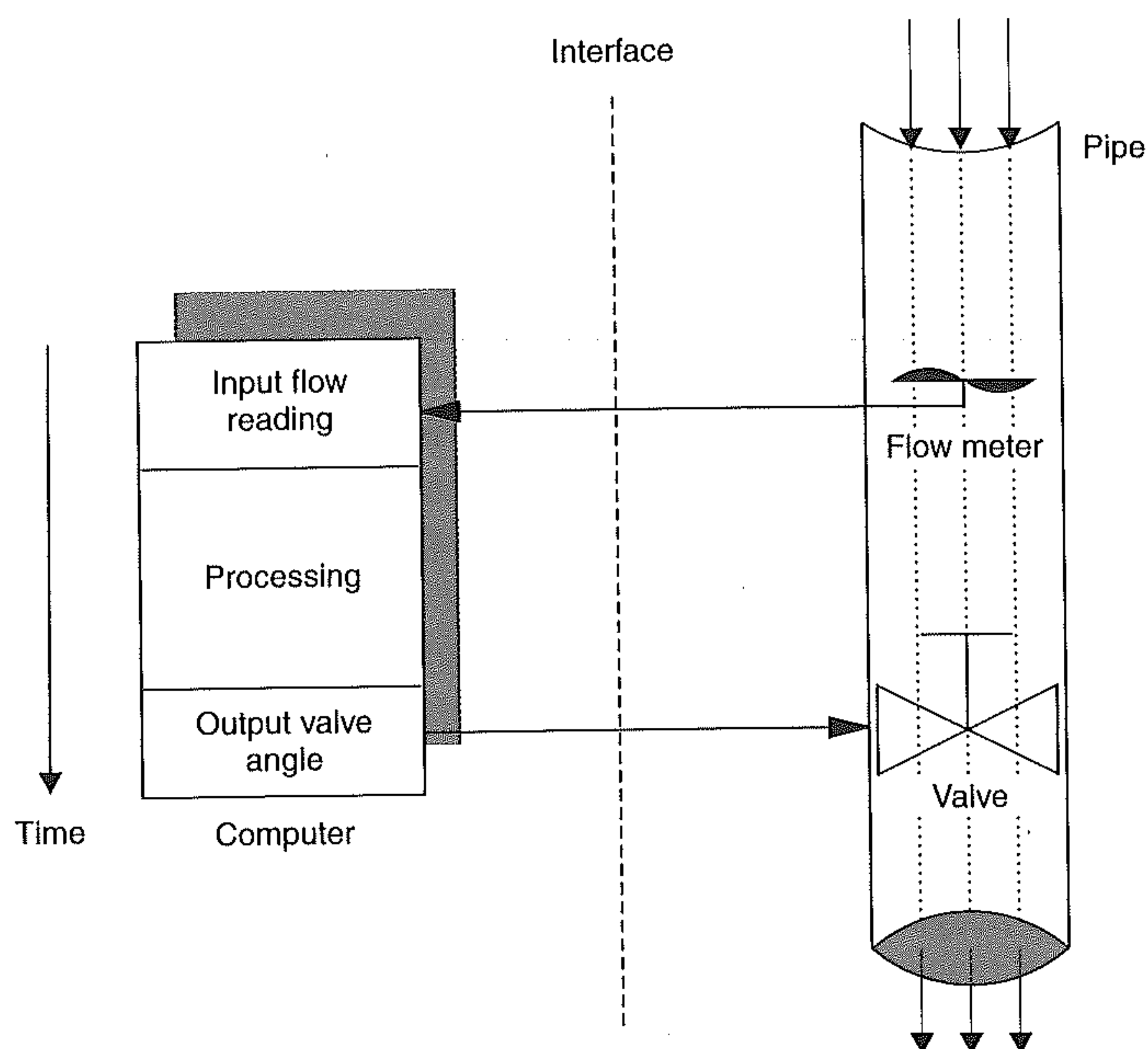


Figure 1.1 A fluid control system.

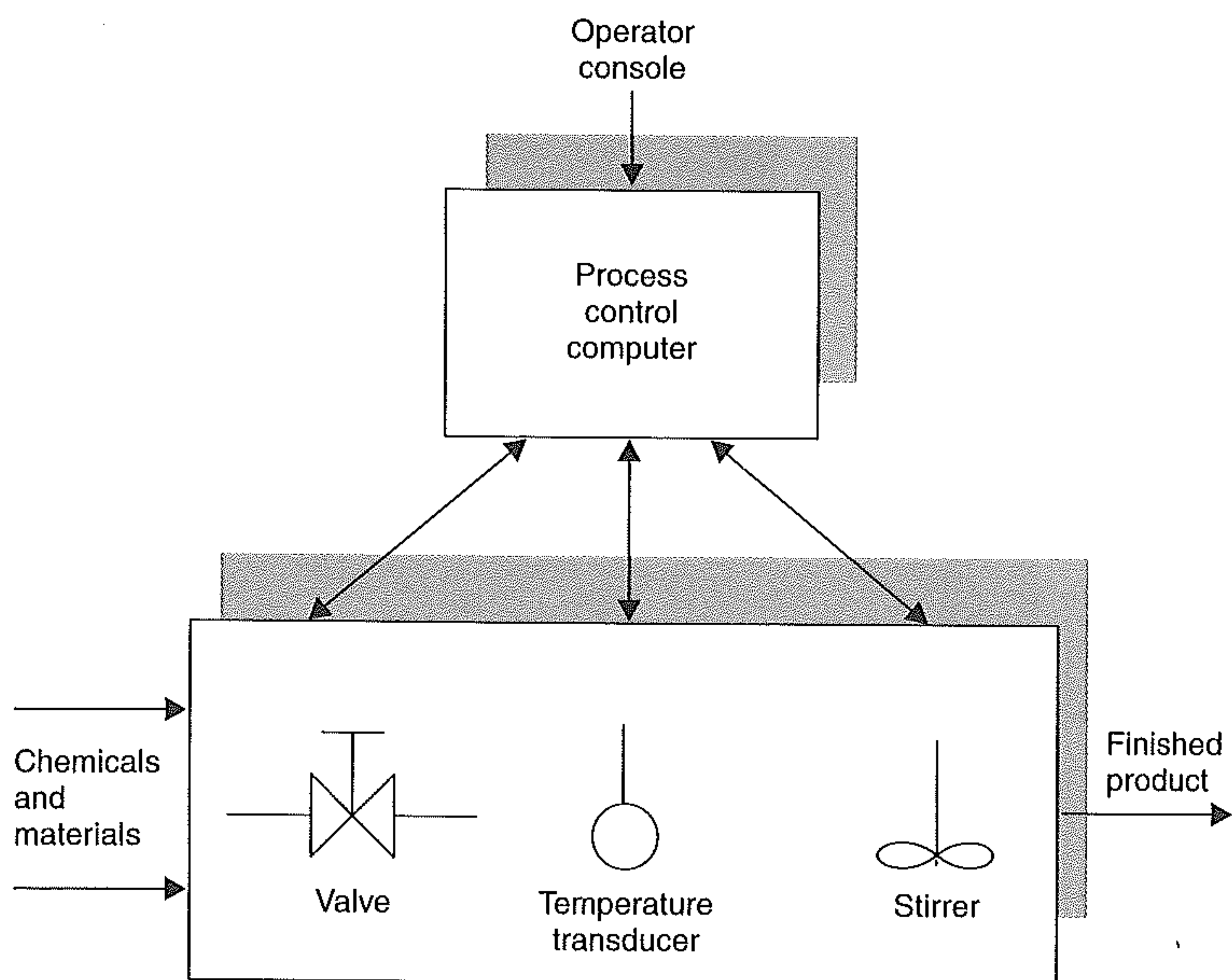


Figure 1.2 A process control system.

the pipe is not to become overloaded. Note that the actual response may involve quite a complex computation in order to calculate the new valve angle.

This example shows just one component of a larger control system. Figure 1.2 illustrates the role of a real-time computer embedded in a complete process control environment. The computer interacts with the equipment using sensors and actuators. A valve is an example of an actuator, and a temperature or pressure transducer is an example of a sensor. (A transducer is a device that generates an electrical signal that is proportional to the physical quantity being measured.) The computer controls the operation of the sensors and actuators to ensure that the correct plant operations are performed at the appropriate times. Where necessary, analog-to-digital (and digital-to-analog) converters must be inserted between the controlled process and the computer.

1.2.2 Manufacturing

The use of computers in manufacturing has become essential in order that production costs can be kept low and productivity increased. Computers have enabled the integration of the entire manufacturing process from product design to fabrication. It is in the area of production control that embedded systems are best illustrated. Figure 1.3 diagrammatically represents the role of the production control computer in the manufacturing process. The physical system consists of a variety of mechanical devices – such as machine tools, manipulators and conveyor belts – all of which need to be controlled and coordinated by the computer.

A modern manufacturing control system will employ a wide range of robots. These will again need to be controlled and coordinated, but they are also autonomous real-time

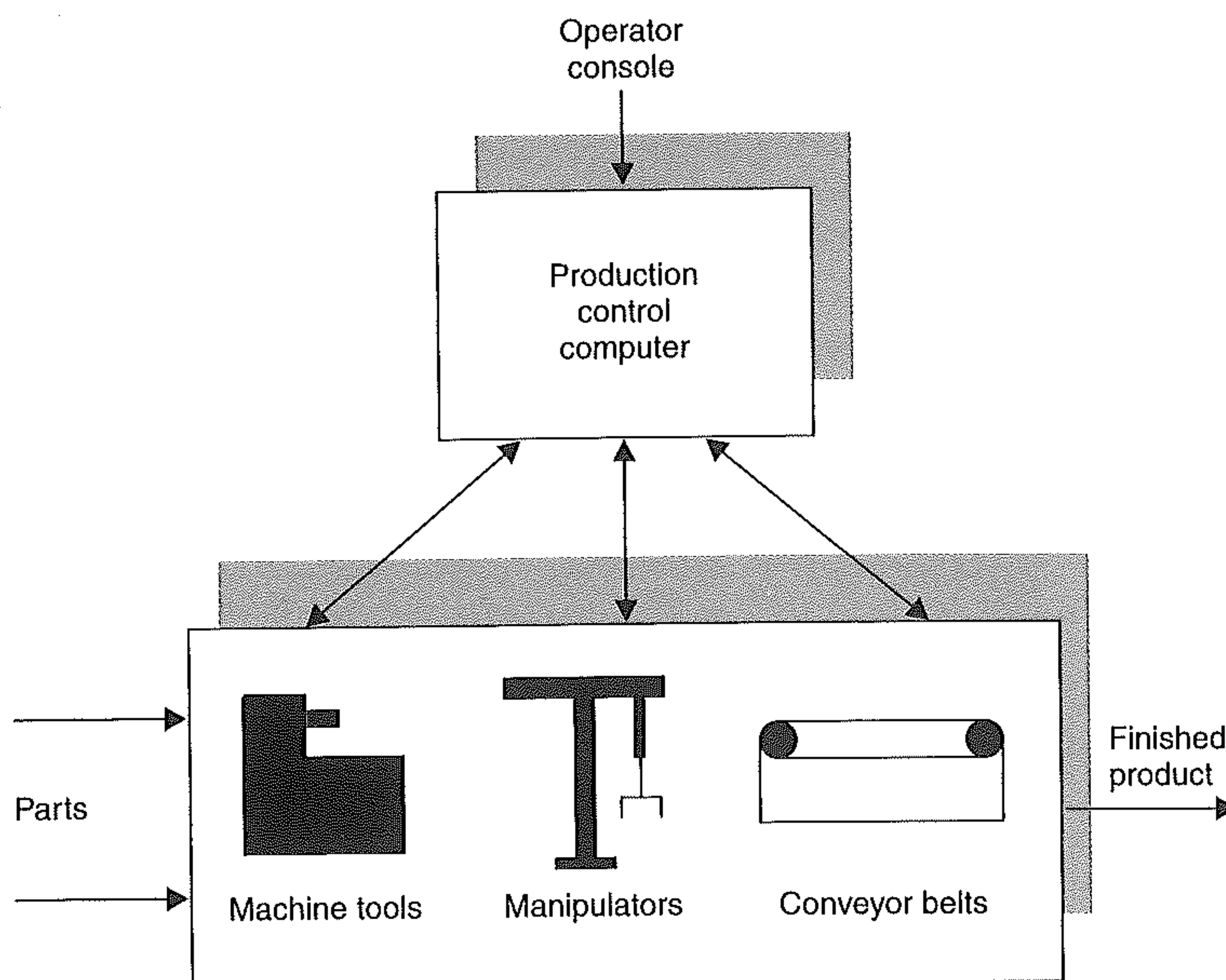


Figure 1.3 A production control system.

systems in their own right. They have large numbers of sensors (for example proximity indicators), many moving parts that need controlling and often vision subsystems that require considerable computational power. When mobile robots and humans operate in the same physical space then there are considerable safety issues that dictate that at least part of the robots' functions are hard real-time.

1.2.3 Communication, command and control

Although **communication, command and control** is a military term, there is a wide range of disparate applications which exhibit similar characteristics; for example, airline seat reservation, medical facilities for automatic patient care, air traffic control, remote bank accounting and large-scale manufacturing plants. Each of these time-aware systems consists of a complex set of policies, information gathering devices and administrative procedures which enable decisions to be supported, and provide the means by which they can be implemented. Often, the information gathering devices and the instruments required for implementing decisions are distributed over a wide geographical area. Figure 1.4 diagrammatically represents such a system.

1.2.4 A typical embedded real-time system

In each of the examples shown above, the computer is interfaced directly to physical equipment in the real world, and is reacting to changes in this environment. In order to control these real-world devices, the computer will need to sample the measurement devices at regular intervals (i.e. periodic activities); a real-time clock is therefore required.

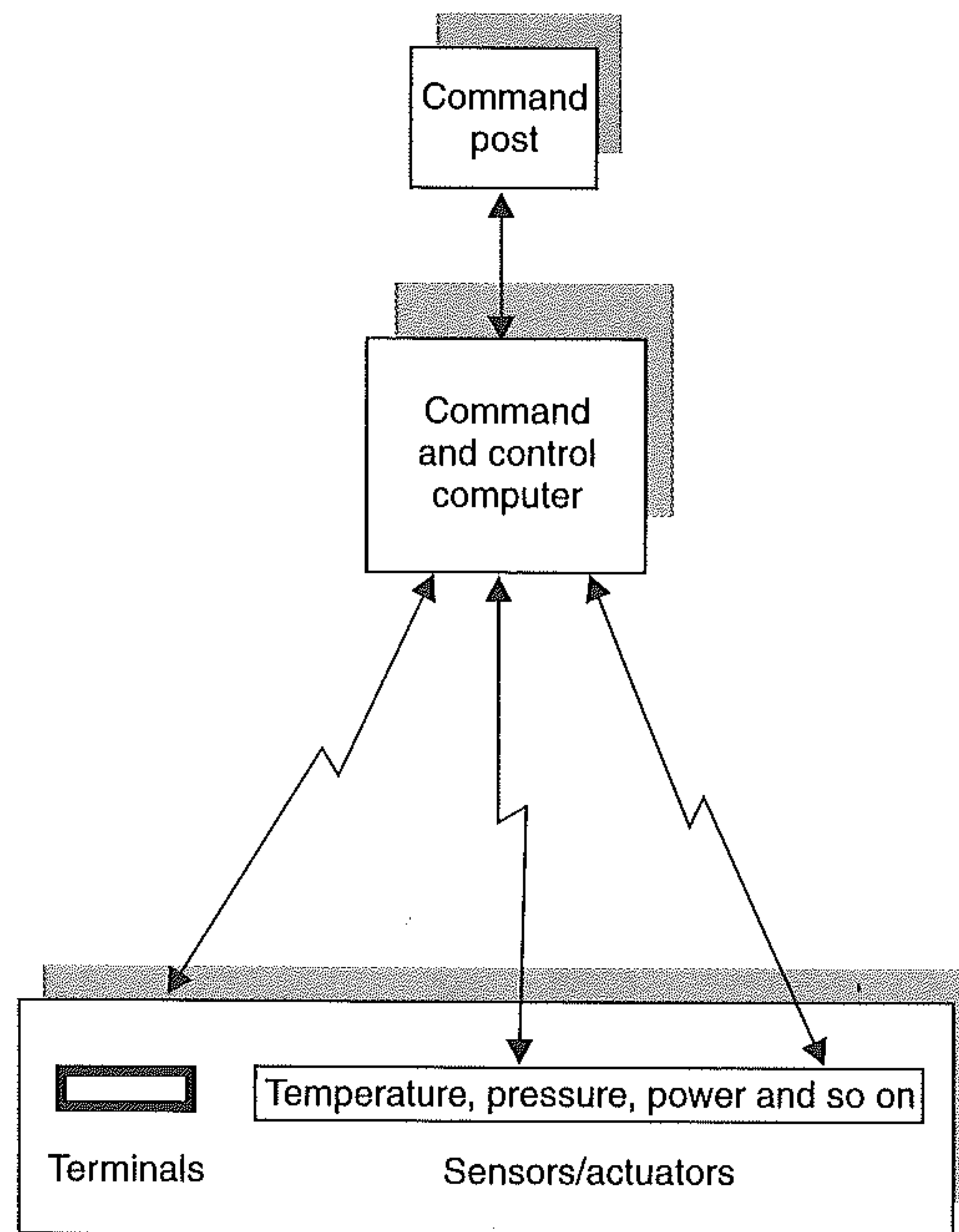


Figure 1.4 A command and control system.

Usually there is also an operator's console to allow for manual intervention. The human operator is kept constantly informed of the state of the system by displays of various types, including graphical ones.

Records of the system's state changes are also kept in a database which can be interrogated by the operators, either for a post mortem (in the case of a system crash), or to provide information for administrative purposes. Indeed, this information is increasingly being used to support decision making in the day-to-day running of systems. For example, in the chemical and process industries, plant monitoring is essential for maximizing economic advantages rather than simply maximizing production. Decisions concerning production at one plant may have serious repercussions for other plants at remote sites, particularly when the products of one process are being used as raw material for another.

A typical large embedded real-time computer system can, therefore, be represented by Figure 1.5. The software which controls the operations of the system can be written in modules which reflect the physical nature of the environment. Usually there will be a module which contains the algorithms necessary for physically controlling the devices; a module responsible for recording the system's state changes; a module to retrieve and display those changes; and a module to interact with the operator.

1.2.5 Multi-media systems

Entertainment systems such as radios, televisions, stereo systems, video systems and games of various kinds are all real-time systems in which the temporal requirements are

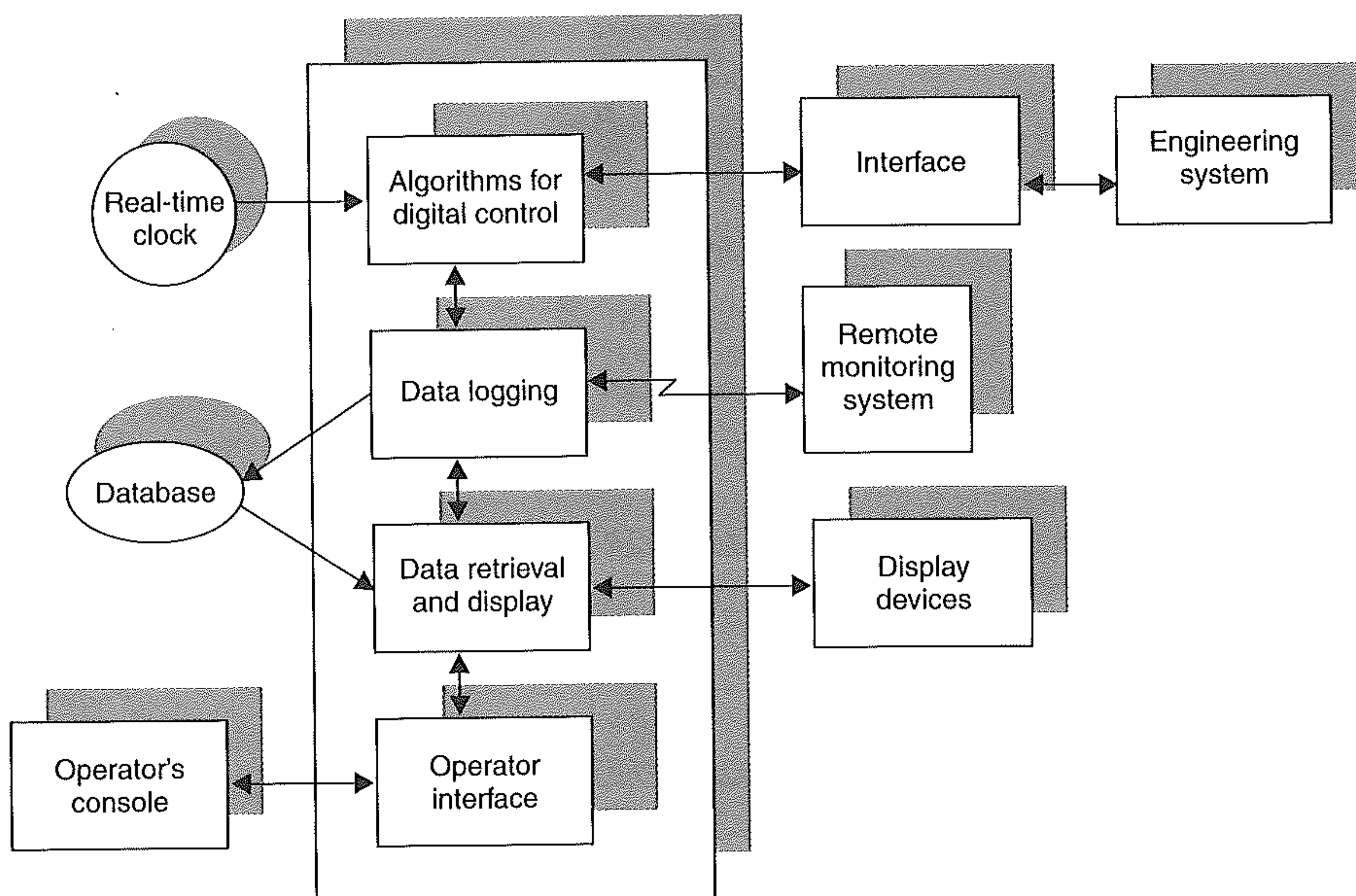


Figure 1.5 A typical embedded system.

determined by the perception and cognition of human users. There are clear requirements on visual frame rates, sound/picture synchronization and response times (to, for example, joystick movements) that must be satisfied if the quality of the human experience is not to be noticeably compromised. Multi-media systems in which many of these entertainment media are integrated with each other and with services such as phone communications, internet searches, email interactions, news updates, and the streaming of audio and video content from remote servers pose significant challenges for developers. There is an economic imperative to manufacture these systems at minimum cost, but the quality of the user experience must be maintained – and this means that the temporal requirements have to be central to the development process. From the design of special purpose on-chip units (for example, video decoders) to the choice of communication subsystems and the implementation of the software control components, the need to manipulate significant volumes of digital data within tight timing deadlines is the key engineering challenge.

The development of these home-based multi-media systems is one noticeable application area for real-time systems, but such fixed systems are not the only commodities under development. There is an increasing need to allow devices to be mobile whilst still linked to communication services. The mobile phone is evolving into a general-purpose essential accessory that supports most of the technologies of the fixed system together with new emerging services such as providing a means of paying for goods in shops, an aid to personal security and/or surveillance and of course a camera and general-purpose recording device. The one single defining property of all mobile systems is that they are indeed *mobile* – they rely on batteries for power. And batteries have finite life which adds to the challenges of developing these complex real-time systems.

1.2.6 Cyber-physical systems

A relatively new application area for real-time systems comes from the linking of digital information systems, such as that furnished by the internet, and real-time data collection typified by networks of sensors. This coupling of what is usually called the cyber world with the physical world has naturally led to the term *cyber-physical systems*. An example of a cyber-physical service is an adaptive navigation aid. Here there is a link between a detailed digitalized (and static) road map and traffic flow information that is being sensed at key locations on the roads. This traffic information is clearly real-time and dynamic. The navigational aid can potentially use sophisticated traffic flow models to predict future problems and advise as to the least congested route. This advice will diminish in usefulness as the data on which it is based becomes stale. Hence the need to update the advice as new data becomes available.

Cyber-physical systems may operate over wide areas, perhaps even globally, involve many computational elements and communication services, and have access to enormous volumes of real-time data. In the traffic system, consider the quantity of data generated if all junctions on all roads in Europe (or the USA or China) are sensing traffic flow (in all directions) and feeding this data into the internet. The sensing component of the cyber-physical system may be quite simple, for example a temperature sensor, or be a significant entity in its own right, such as a camera permanently viewing some scene. The local sensory system must, in this case, decide whether raw data or a processed (simpler) form of the data is fed into the digital world.

A cyber-physical system is likely to have real-time requirements that range from the millisecond level for the sensing activity to minutes or even hours for the applications that are built upon the amalgamation of the dynamic and static data. It should be clear to all readers, who have presumably spent time ‘on the net’, that the current internet with its simple protocols and architecture cannot guarantee real-time behaviour. Future enhancements (or alternatives) to the internet are, however, likely to respond to the challenges of cyber-physical systems and provide QoS (Quality of Service) that does provide real-time guarantees – even if the guarantees are not absolute (hard) but have a high level of probability associated with them.

1.3 Characteristics of real-time systems

A real-time system possesses many special characteristics (either inherent or imposed) which are identified in the following sections. Clearly, not all real-time systems will exhibit all these characteristics; however, any general-purpose language (and operating system) which is to be used for the effective programming of real-time systems must have facilities which support these characteristics.

1.3.1 Real-time facilities

Response time is crucial in any embedded system. Unfortunately, it is very difficult to design and implement systems which will guarantee that the appropriate output will be generated at the appropriate times under all possible conditions. It is often impossible to do this, and make full use of all computing resources at all times. For this reason, real-time systems are usually constructed using processors with considerable spare capacity,

thereby ensuring that ‘worst-case behaviour’ does not produce any unwelcome delays during critical periods of the systems’ operations.

Given adequate processing power, language and run-time support are required to enable the programmer to:

- specify times at which actions are to be performed;
- specify times by which actions are to be completed;
- support repeating (periodic or aperiodic) work;
- control (i.e. bound) the jitter on input and output operations;
- respond to situations where not all of the timing requirements can be met;
- respond to situations where the timing requirements are changed dynamically.

These are called real-time control facilities. They enable the program to synchronize with time itself. For example, with direct digital control algorithms it is necessary to sample readings from sensors at certain periods of the day, for example, 2 p.m., 3 p.m. and so on, or at regular intervals, for instance, every 5 seconds (with control systems, sample rates can vary from a few hundred hertz to several hundred megahertz). As a result of these readings, other actions will need to be performed. In an electric power station, it is necessary at 6 p.m. on Monday to Friday each week to increase the supply of electricity to domestic consumers. This is in response to the peak in demand caused by families returning home from work, turning on lights, cooking dinner and so on. In recent years in the UK, the demand for domestic electricity reaches a peak immediately after high-profile sporting events, when millions of viewers leave their living rooms, turn on lights in the kitchen and switch on the kettle in order to make a cup of tea or coffee.

An example of a dynamic change to the timing requirements of a system can be found in an aircraft flight control system. If an aircraft has experienced depressurization, there is an immediate need for all computing resources to be given over to handling the emergency. More normally, the moves from taxiing to taking off to climbing and then to cruising all involve changes to the basic operation of the flight control system. These changes, which are known generally as **mode changes**, also have consequences for the temporal characteristics of the executing software.

In order to meet response times, it is necessary for a system’s behaviour to be predictable. Chapters 9–12 consider the facilities and techniques used to obtain predictable program behaviour.

1.3.2 Concurrent control of separate system components

An embedded system will tend to consist of computers and several coexisting external elements with which the computer programs must interact simultaneously. It is the very nature of these external real-world elements that they exist in parallel. In our typical embedded computer example, the program has to interact with an engineering system (which will consist of many parallel activities such as robots, conveyor belts, sensors, actuators and so on) and the computer’s display devices, the operator’s console, the database and the real-time clock. Fortunately, the speed of a modern computer is such that usually these actions may be carried out in sequence but give the illusion of being simultaneous. In some embedded systems, however, this may not be the case, for example

where the data is to be collected and processed at various geographically distributed sites, or where the response time of the individual components cannot be met by a single computer. In these cases, it is necessary to consider distributed and multiprocessor embedded systems.

A major problem associated with the production of software for systems which exhibit concurrency is how to express that concurrency in the structure of the program. One approach is to leave it all up to the programmer who must construct his/her system so that it involves the cyclic execution of a program sequence to handle the various concurrent tasks. There are several reasons, however, why this is inadvisable.

- It complicates the programmer's already difficult task and involves him or her in considerations of structures which are irrelevant to the control of the tasks in hand.
- The resulting program is more obscure and inelegant.
- It makes proving program correctness more difficult.
- It makes decomposition of the problem more complex.
- Parallel execution of the program on more than one processor is much more difficult to achieve.
- The placement of code to deal with faults is more problematic.

Older real-time programming languages, for example, RTL/2 and Coral 66, relied on operating system support for concurrency; and C is usually associated with Unix, Linux or POSIX. However, the more modern languages, such as Ada and Java, have direct support for general concurrent programming.

Although concurrency is a fundamental characteristic of real-time systems, different types of systems need different facilities. For reactive control systems with hard timing constraints, but relatively straightforward behaviour, it is sufficient to constrain each concurrent activity to be of the form:

```
input data
required computations with no external interactions
output data
```

For more complicated systems, the input and output activities cannot be separated from the computational part. Only during the computation activity itself will it be possible to determine the external data that is needed (for example from another activity, a shared database or from some interface into the environment).

In Chapters 4, 5 and 6, various models of concurrent programming are considered in detail. Attention is then focused, in the following two chapters, on achieving reliable communication and synchronization between concurrent processes in the presence of design errors.

1.3.3 Low-level programming

The nature of embedded systems requires the computer components to interact with the external world. They need to monitor sensors and control actuators for a wide variety of real-world devices. These devices interface to the computer via input and output registers, and their operational requirements are device- and computer-dependent. Devices may

also generate interrupts to signal to the processor that certain operations have been performed or that error conditions have arisen.

In the past, interfacing to devices has either been left under the control of the operating system, or has required the application programmer to resort to assembly language inserts to control and manipulate the registers and interrupts. Nowadays, because of the variety of devices and the time-critical nature of their associated interactions, their control must often be direct, and not through a layer of operating system functions. Furthermore, reliability requirements argue against the use of low-level programming techniques.

Since real-time systems are time-critical, efficiency of implementation will be more important than in other systems. It is interesting that one of the main benefits of using a high-level language is that it enables the programmer to abstract away from implementation details, and to concentrate on solving the problem at hand. Unfortunately, the embedded computer systems programmer cannot afford this luxury. He or she must be constantly concerned with the cost of using particular language features. For example, if a response to some input is required within a microsecond there is no point in using a language feature whose execution takes a millisecond!

It was noted earlier that a significant number of embedded systems are now mobile and rely on batteries for power. Efficient implementation can reduce the load on the processor, and memory, and lead to lowering of processor speed and the subsequent reduction in battery usage. Typically, halving of processor speed leads to a quadrupling of the life of the battery.

In Chapter 14, the facilities provided by real-time programming languages which enable the specification of device registers and interrupt control will be considered. The role of the execution environment in providing efficient and predictable implementations will also be examined.

1.3.4 Support for numerical computation

As was noted earlier, many real-time systems involve the control of some engineering activity. Figure 1.6 exemplifies a simple control system. The controlled entity, the plant, has a vector of output variables, y , that change over time, hence $y(t)$. These outputs are compared with the desired (or reference) signal $r(t)$ to produce an error signal, $e(t)$. The controller uses this error vector to change the input variables to the plant, $u(t)$. For a very simple system, the controller can be an analog device working on a continuous signal.

Figure 1.6 illustrates a feedback controller. This is the most common form, but feed-forward controllers are also used. In order to calculate what changes must be made

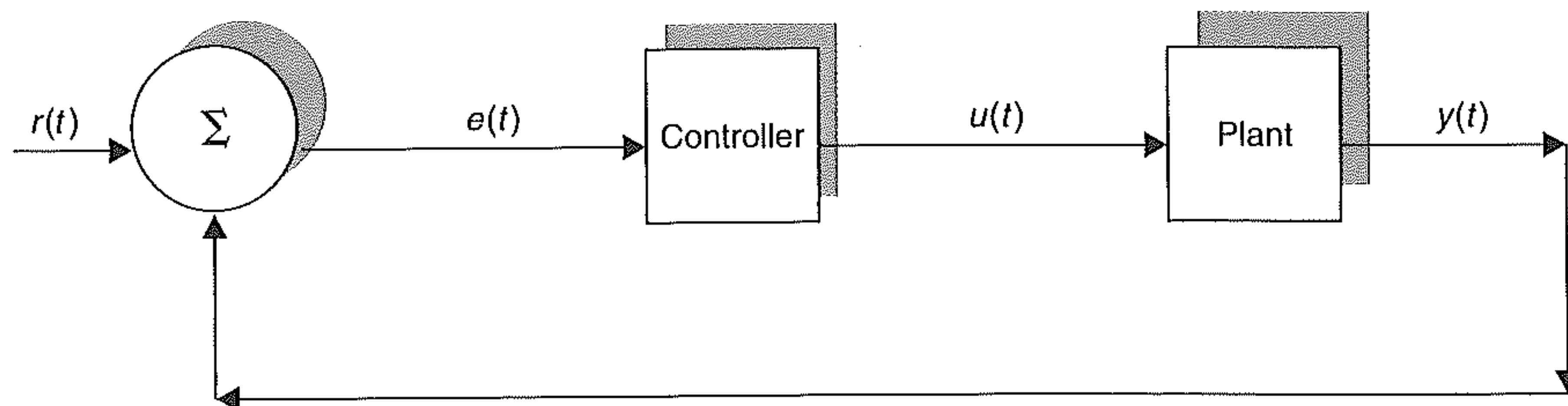


Figure 1.6 A simple controller.

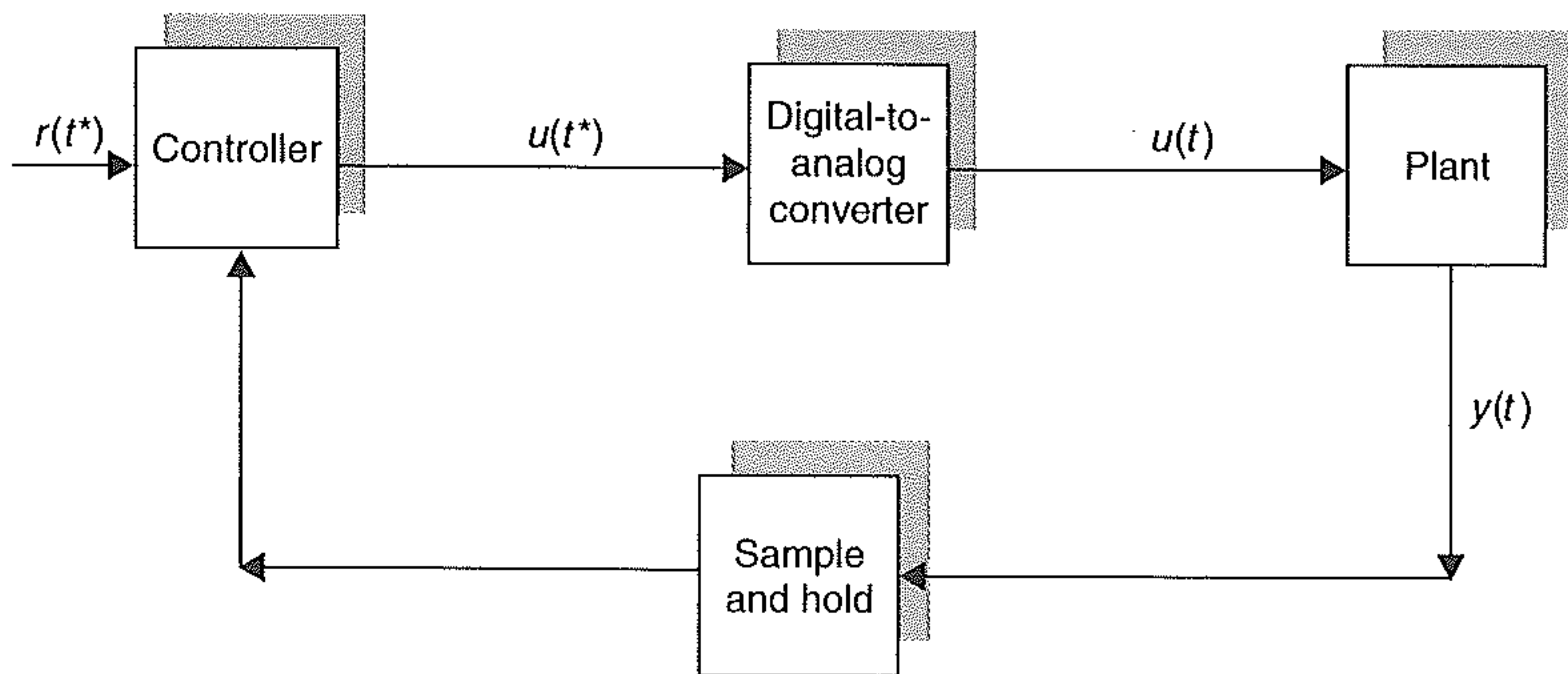


Figure 1.7 A simple computerized controller.

to the input variables, so that a desirable effect on the output vector takes place, it is necessary to have a mathematical model of the plant. The derivation of these models is the concern of the distinct discipline of control theory. Often a plant is modelled as a set of first-order differential equations. These link the output of the system with the internal state of the plant and its input variables. Changing the output of the plant involves solving these equations to give required input values. Most physical systems exhibit inertia, so that change is not instantaneous. A real-time requirement to move to a new set point within a fixed time period will add to the complexity of the manipulations needed, both to the mathematical model and to the physical system. The fact that, in reality, linear first-order equations are only an approximation to the actual characteristics of the system also presents complications.

Because of these difficulties, the complexity of the model, and the number of distinct (but not independent) inputs and outputs, most controllers are implemented as digital computers. The introduction of a digital component into the system changes the nature of the control cycle. Figure 1.7 is an adaptation of the earlier model. Items marked with a * are now discrete values; the sample and hold operation being carried out by an analog-to-digital converter, both converters being under the direct control of the computer.

Within the computer, the differential equations can be solved by numerical techniques, although the algorithms themselves need to be adapted to take into account the fact that plant outputs are now being sampled. The design of control algorithms is a topic outside the scope of this book; the implementation of these algorithms is, however, of direct concern. They can be mathematically complex and require a high degree of precision. A fundamental requirement of a real-time programming language, therefore, is the ability to manipulate real or floating-point numbers. Fortunately most engineering languages do provide the necessary abstractions in this area. There are standards for floating-point arithmetic that fully specify the required output of floating-point computations even when there are overflows, underflow, etc.

1.3.5 Large and complex

It is often said that most of the problems associated with developing software are those related to size and complexity. Writing small programs presents no significant problems

as they can be designed, coded, maintained and understood by a single person. If that person leaves the company or institution using the software, then someone else can learn the program in a relatively short period of time. Indeed, for these programs, there is an *art* or *craft* to their construction and *small is beautiful*.

Unfortunately, not all software exhibits this most desirable characteristic of smallness. Lehman and Belady (1985), in attempting to characterize large systems, reject the simple and perhaps intuitive notion that largeness is simply proportional to attributes such as: the number of instructions, lines of code or modules making up a program and to algorithmic complexity. Instead, they relate largeness to **variety**, and the degree of largeness to the amount of variety. Traditional indicators such as number of instructions and development effort are, therefore, just symptoms of variety.

The variety is that of needs and activities in the real world and their reflection in a program. But the real world is continuously changing. It is evolving. So too are, therefore, the needs and activities of society. Thus large programs, like all complex systems, must continuously evolve.

Embedded systems by their definition must respond to real-world events. The variety associated with these events must be catered for; the programs will, therefore, tend to exhibit the undesirable property of largeness. Inherent in the above definition of largeness is the notion of **continuous change**. The cost of redesigning or rewriting software to respond to the continuously changing requirements of the real world is prohibitive. Therefore real-time systems undergo constant maintenance and enhancements during their lifetimes. They must be extensible.

Although real-time software is often complex, features provided by real-time languages and environments enable these complex systems to be broken down into smaller components which can be managed effectively. The use of abstract data types, classes and objects, generic components, Application Program Interfaces (APIs) and interfaces, and separate compilation are all language features that engineering languages provide to manage this software complexity.

1.3.6 Extremely reliable and safe

The more society relinquishes control of its vital functions to computers, the more imperative it becomes that those computers do not fail. The failure of a system involved in automatic fund transfer between banks can lead to millions of dollars being lost irretrievably; a faulty component in electricity generation could result in the failure of a vital life-support system in an intensive care unit; and the premature shutdown of a chemical plant could cause expensive damage to equipment or environmental harm. These somewhat dramatic examples illustrate that computer hardware and software must be reliable and safe. Even in hostile environments, such as those found in military applications, it must be possible to design and implement systems that will fail only in a controlled way. Furthermore, where operator interaction is required, care must be taken in the design of the interface in order to minimize the possibility of human error.

The sheer size and complexity of real-time systems exacerbate the reliability problem; not only must expected difficulties inherent in the application be taken into account, but also those introduced by faulty software design.

In Chapters 2 and 3, the problems of producing reliable and safe software will be considered along with the facilities that languages have introduced to cope with both expected and unexpected error conditions. The issue is examined further in Chapters 7 and 13.

1.3.7 Structure of the book

The characteristics of real-time systems outlined in these short sections introduce the topics that are covered in this book. In Chapters 2 and 3, the problems of producing reliable and safe software will be considered along with the facilities that languages have introduced to cope with both expected and unexpected error conditions. The issue is examined further in Chapters 7 and 13.

In Chapters 4, 5 and 6, various models of concurrent programming are considered in detail. Attention is then focused, in the following two chapters, on achieving reliable communication and synchronization between concurrent processes in the presence of design errors. Support for real-time abstractions is discussed in Chapters 9, 10 and 11, together, in Chapter 12, with the language facilities that assist in the scheduling of time-critical operations. The techniques used to detect and recover from timing failures are considered in Chapter 13.

In Chapter 14, the facilities provided by real-time programming languages which enable the specification of device registers and interrupt control are considered, along with the role of the execution environment in providing efficient and predictable implementations.

Finally in Chapter 15 a case study, that brings together many of the key issues of the book, is developed.

The remainder of this introductory chapter outlines a number of other important issues whose detailed consideration is nevertheless beyond the scope this book. In particular the development cycle (i.e. requirements specification, design, implementation and testing) is introduced and a number of general language topics are summarized. These discussions provide the backdrop to the detailed treatments in the following chapters and help deal with the size and complexity problems inherent in the development of most real-time systems.

1.4 Development cycle for real-time systems

Clearly, the most important stage in the development of any real-time system is the generation of a consistent design that satisfies an authoritative specification of requirements. In this, real-time systems are no different from other computer applications, although their overall scale often generates quite fundamental design problems. The discipline of software engineering is now widely accepted as the focus for the development of methods, tools and techniques aimed at ensuring that the software production process is manageable, and that reliable and correct programs are constructed. It is assumed here that readers are familiar with the basic tenets of software engineering and consideration is thus restricted to the particular problems and requirements furnished by real-time embedded systems. Even within this restriction, it is not possible to give a comprehensive account of the many design methodologies proposed. Issues of design *per se* are not the main focus of attention in this book. Rather, the investigation of language and operating

system primitives, which allow designs to be realized, is the central theme. Within this context, the languages Ada, (Real-Time) Java and C (with Real-Time POSIX) will be considered in detail. Readers should consult the further reading list at the end of this chapter for additional material on the design process.

Although almost all design approaches are top-down, they are built upon an understanding of what is feasible at lower levels. In essence, all design methods involve a series of transformations from the initial statement of requirements to the executing code. This section gives a brief overview of some of the typical stages that are passed through on this route, that is:

- **requirements specification** – during which an authoritative specification of the system's required functional and meta-functional behaviour is produced;
- **architectural design** – during which a top-level description of the proposed system is developed;
- **detailed design** – during which the complete system design is specified;
- **coding** – during which the system is implemented;
- **testing** – during which the efficacy of the system is tested.

As different activities are isolated, notations are required that enable each stage to be documented. Transformations from one stage to another are, therefore, nothing more than translations from one notation to another. For example, a compiler produces executable code from source code that is expressed in a programming language. Unfortunately, other translations (further up the design hierarchy) are less well defined; usually because the notations employed are too vague and imprecise, and cannot fully capture the semantics of the requirements or of the design.

Linked to notations are the models that can be expressed in these languages. The current emphasis on development approaches is focused on what is called **model driven architectures** (MDA). Here formal notations are used to develop models of the systems that can then be subject to verification. Techniques such as model-checking and mechanical proof are used to increase the designer's confidence that the systems once fully implemented will behave as expected in both the functional and temporal domains. Advocates of the MDA approach argue that it will be possible to automatically generate the executable code of the implementation from these higher-level models. This may well be the right approach for the future for relatively straightforward systems. However, the current state of these MDA approaches is not sufficiently advanced for general real-time systems to be developed by code-generation techniques. Hence in this book the focus is on the abstractions and facilities provided by real-time programming languages.

1.4.1 Requirement specification

Almost all computing projects start with an informal description of what is desired. This should then be followed by an extensive analysis of requirements. It is at this stage that the functionality of the system is defined. In terms of specific real-time factors, the temporal behaviour of the system should be made quite explicit, as should the reliability requirements and the desired behaviour of the software in the event of component failure. The requirements phase will also define which acceptance tests should apply to the software.

In addition to the system itself, it is necessary to build a model of the environment of the application. It is a characteristic of real-time systems that they have important interactions with their environment. Hence such issues as maximum rate of interrupts, maximum number of dynamic external objects (for example, aeroplanes in an air traffic control system) and failure modes are all important.

The analysis phase provides an authoritative specification of requirements. It is from this that the design will emerge. There is no more critical phase in the software life cycle, and yet natural language documents are still the normal notation for this specification. Nevertheless, formal methods can be applied to these specifications and are increasing being required in safety-critical applications.

1.4.2 Design activities

The design of a large embedded system cannot be undertaken in one exercise. It must be structured in some way. To manage the development of complex real-time systems, two complementary approaches are often used: decomposition and abstraction. Together, they form the basis of most software engineering methods. Decomposition, as its name suggests, involves the systematic breakdown of the complex system into smaller and smaller parts until components are isolated that can be understood and engineered by individuals or small groups. At each level of decomposition, there should be an appropriate level of description and a method of documenting (expressing) this description. Abstraction enables the consideration of detail, particularly that appertaining to implementation, to be postponed. This allows a simplified view of the system and of the objects contained within it to be taken, which nevertheless still contains the essential properties and features. The use of abstraction and decomposition pervades the entire engineering process and has influenced the design of real-time programming languages and associated software design methods.

If a formal notation is used for the requirement specification then top-level designs may use the same notation and can thus be proven to meet the specification. Many structured notations are, however, advocated either to fill out the top-level design or to replace the formal notation altogether. Indeed, a structured top-level design may, in effect, be the authoritative specification of requirements.

The hierarchical development of software leads to the specification and subsequent development of program subcomponents. The needs of abstraction dictate that these subcomponents should have well-defined roles, and clear and unambiguous interconnections and interfaces. If the specification of the entire software system can be verified just in terms of the specification of the immediate subcomponents then decomposition is said to be **compositional**. This is an important property when formally analysing programs.

Sequential programs are particularly amenable to compositional methods, and a number of techniques have been used to encapsulate and represent subcomponents. Simula introduced the significant **class** construct. More recently *object-oriented* languages, such as C++, Java and Eiffel, have emerged to build upon the class construct. Ada uses a combination of packages, type extensions and interfaces to support object-oriented programming.

Objects, while providing an abstract interface, require extra facilities if they are to be used in a concurrent environment. Typically, this involves the addition of some

form of task (process). The **task** abstraction is, therefore, the abstraction on which this book will focus. In Chapter 4, the notion of task is introduced, Chapter 5 then considers shared-variable based tasks interaction. A more controlled and abstract interface is, however, provided by message-based process communication. This is discussed in Chapter 6.

Both object and task abstractions are important in the design and implementation of reliable embedded systems. These forms of encapsulation lead to the use of modules with well-defined (and abstract) interfaces. From the definition of modules, more sizeable *components* can be defined that may even be re-usable in subsequent designs. But how should a large system be decomposed into components and modules? To a large extent, the answer to this question lies at the heart of all software design activities. Cohesion and coupling are two metrics that are often used to describe the relationships between entities within a design (in the following, the term ‘module’ is used for a distinct software entity).

Cohesion is concerned with how well a module holds together – its internal strength. Allworth and Zobel (1987) give six measures of cohesion that range from the very poor to the strong.

- **Coincidental** – elements of the module are not linked other than in a very superficial way; for example, written in the same month.
- **Logical** – elements of the module are related in terms of the overall system, but not in terms of the actual software; for example all output device drivers.
- **Temporal** – elements of the module are executed at similar times; for example, start-up routines.
- **Procedural** – elements of the module are used together in the same section of the program; for example, user interface components.
- **Communicational** (*sic*) – elements of the module work on the same data structure; for example, algorithms used to analyse an input signal.
- **Functional** – elements of the module work together to contribute to the performance of a single system function; for example, the provision of a distributed file system.

Coupling, by comparison, is a measure of the interdependence of program modules. If two modules pass control information between them, they are said to possess high (or tight) coupling. Alternatively, the coupling is loose if only data is communicated. Another way of looking at coupling is to consider how easy it would be to remove a module (from a completed system) and replace it with an alternative one.

Within all design methods, a good decomposition is one that has *strong* cohesion and *loose* coupling. This principle is equally true in sequential and concurrent programming domains.

It was noted earlier that most real-time practitioners advocate the use of object and task (process) abstractions. Formal techniques do exist that enable concurrent time-constrained systems to be specified and analysed. Nevertheless, these techniques are not yet sufficiently mature to constitute ‘tried and tested’ design methods. Rather, the real-time industry uses, at best, structured methods and software engineering approaches that are applicable to all information processing systems. They do not give specific

support to the real-time domain, and they lack the richness that is needed if the full power of implementation languages is to be exploited.

1.4.3 Testing and simulation

With the high reliability requirements that are the essence of most real-time systems, it is clear that testing must be extremely stringent. A comprehensive strategy for testing involves many techniques, most of which are applicable to all software products. It is, therefore, assumed that the reader is familiar with these techniques.

The difficulty with real-time concurrent programs is that the most intractable system errors are usually the result of subtle interactions between tasks. Often the errors are also time-dependent and will only manifest themselves in rare states. Murphy's Law dictates that these rare states are also crucially important and only occur when the controlled system is, in some sense, critical.

Testing is, of course, not restricted to the final assembled system. The decomposition incorporated in the design and manifest within program modules (including tasks) forms a natural architecture for component testing. Of particular importance (and difficulty) within real-time systems is that not only must correct behaviour in a correct environment be tested, but dependable behaviour in an arbitrarily incorrect environment must be catered for. All error recovery paths must be exercised and the effects of simultaneous errors investigated.

To assist in any complex testing activity, a realistic test bed presents many attractions. For software, such a test environment is called a simulator.

A simulator is a program which imitates the actions of the engineering system in which the real-time software is embedded. It simulates the generation of interrupts and performs other I/O actions in real-time. Using a simulator, abnormal as well as 'normal' system behaviour can be created. Even when the final system has been completed, certain error states may only be safely experimented with via a simulator. The meltdown of a nuclear reactor is an obvious example.

Simulators are able to reproduce accurately the sequence of events expected in the real system. In addition, they can repeat experiments in a way that is usually impossible in a live operation. However, to faithfully recreate simultaneous actions it may be necessary to have a very powerful computational platform. And even then with very complicated applications it may not be possible to build an appropriate emulation of the system.

Simulators are clearly non-trivial and expensive systems to develop. They may even require special hardware. In the NASA Space Shuttle project, the simulators cost more than the real-time software itself. This money turned out to be well spent, with many system errors being found during hours of simulator 'flight'.

1.4.4 Postscript

In many ways, this discussion on design issues has been a divergent one. It has introduced more problem areas and engineering issues than can possibly be tackled in just one book. This broad sweep across the 'design process' is aimed at setting the rest of the book in context. By now focusing on language issues and programming activities, the reader will be able to understand the 'end product' of design, and judge to what extent current methodologies, techniques and tools are appropriate.

1.5 Languages for programming real-time systems

An important plateau between the top-level requirements specification and the executing machine code is the programming language. The development of implementation languages for real-time systems is the central theme of this book. Language design is still a very active research area. Although systems design should lead naturally into implementation, the expressive power of most modern languages is not matched by current design methodologies. Only by understanding what is possible at the implementation stage can appropriate design approaches be developed.

It is possible to identify three classes of programming languages which are, or have been, used in the development of real-time systems. These are assembly languages, sequential systems implementation languages and high-level concurrent languages. These types of languages will shortly be reviewed, but first some general language design criteria will be introduced.

1.5.1 General language design criteria

Although a real-time language may be designed primarily to meet the requirements of embedded computer system programming, its use is rarely limited to that area. Most real-time languages are also used as general-purpose systems implementation languages for applications such as compilers and operating systems.

Young (1982) lists the following six (sometimes conflicting) criteria as the basis of a real-time language design: security, readability, flexibility, simplicity, portability and efficiency. A similar list also appears in the original requirements for Ada.

Security

The security of a language design is a measure of the extent to which programming errors can be detected automatically by the compiler or language run-time support system. There is obviously a limit to the type and number of errors that can be detected by a language system; for example, errors in the programmer's logic cannot be detected automatically. A secure language must, therefore, be well structured and readable so that such errors can easily be spotted.

The benefits of security include:

- the detection of errors much earlier in the development of a program – generating an overall reduction in cost;
- compile-time checks have no overheads at run-time – a program is executed much more often than it is compiled.

The disadvantage of security is that it may result in a more complicated language with an increase in compilation time and compiler complexity.

Readability

The readability of a language depends on a variety of factors including the appropriate choice of keywords, the ability to define types and the facilities for program

modularization. As Young points out:

the aim is to provide a language notation with sufficient clarity to enable the main concepts of a particular program's operation to be assimilated easily by reading the program's text only, without resort to subsidiary flowcharts and written descriptions. (Young, 1982)

The benefits of good readability include:

- reduced documentation costs;
- increased security;
- increased maintainability.

The main disadvantage is that it usually increases the length of any given program.

Flexibility

A language must be sufficiently flexible to allow the programmer to express all the required operations in a straightforward and coherent fashion. Otherwise, as with older sequential languages, the programmer will often have to resort to operating system commands or machine code inserts to achieve the desired result.

Simplicity

Simplicity is a worthwhile aim of any design, be it of the international space station or a simple calculator. In programming languages, simplicity has the advantages of:

- minimizing the effort required to produce compilers;
- reducing the cost associated with programmer training;
- diminishing the possibility of making programming errors as a result of misinterpretation of the language features.

Flexibility and simplicity can also be related to the **expressive power** (the ability to express the solutions to a wide range of problems) and **usability** (ease of use) of the language.

Portability

A program, to a certain extent, should be independent of the hardware on which it executes. One of the main claims of Java is that programs are compiled once and run anywhere. For a real-time system, this is difficult to achieve (even with the advent of portable binary codes, such as Java Byte Code and ANDF (Venners, 1999; X/Open Company Ltd, 1996)), as a substantial part of any program will normally involve manipulation of hardware resources. However, a language must be capable of isolating the *machine-dependent* part of a program from the *machine-independent* part.

Efficiency

In a real-time system, response times must be guaranteed; therefore the language must allow efficient and predictable programs to be produced. Mechanisms which lead to unpredictable run-time overheads should be avoided. Obviously, efficiency requirements must be balanced against security, flexibility and readability requirements.

1.5.2 Assembly languages

Initially, most real-time systems were programmed in the assembly language of the embedded computer. This was mainly because high-level programming languages were not well supported on most microcomputers and assembly language programming appeared to be the only way of achieving efficient implementations that could access hardware resources.

The main problem with the use of assembly languages is that they are machine-oriented rather than problem-oriented. The programmer can become encumbered with details which are unrelated to the algorithms being programmed, with the result that the algorithms themselves become obscure. This keeps development costs high and makes it very difficult to modify programs when errors are found or enhancements required.

Further difficulties arise because programs cannot be moved from one machine to another but must be rewritten. Also staff must be retrained if they are required to work with other machines.

1.5.3 Sequential systems implementation languages

As computers became more powerful, programming languages more mature, and compiler technology progressed, the advantages of writing real-time software in a high-level language outweighed the disadvantages. To cope with deficiencies in languages like FORTRAN, new languages were developed specifically for embedded programming. In the United States Air Force, for example, Jovial was in common use. In the UK, the Ministry of Defence standardized on Coral 66, and large industrial concerns like ICI standardized on RTL/2. Currently, the C and C++ programming languages are popular.

All these languages have one thing in common – they are sequential. They also tend to be weak in the facilities they provide for real-time control and reliability. As a result of these shortcomings, it is often necessary to rely on operating system support and assembly code inserts.

1.5.4 High-level concurrent programming languages

In spite of the increasing use of application-tailored languages, the production of computer software became progressively more difficult during the 1970s as computer-based systems became larger and more sophisticated. These problems grew to what became known as the *software crisis*. There are several symptoms of this crisis which have been recognized (Booch, 1986).

- **Responsiveness** – production systems which have been automated often do not meet users' needs.

- **Reliability** – software is unreliable and will often fail to perform to its specification.
- **Cost** – software costs are seldom predictable.
- **Modifiability** – software maintenance is complex, costly and error prone.
- **Timeliness** – software is often delivered late.
- **Transportability** – software in one system is seldom used in another.
- **Efficiency** – software development efforts do not make optimal use of the resources involved.

Perhaps one of the best illustrations of the impact of the software crisis can be found in the American Department of Defense's (DoD) search for a common high-order programming language for all its applications. As hardware prices began to fall during the 1970s, the DoD's attention was focused on the rising cost of its embedded software. It estimated that, in 1973, three thousand million dollars were spent on software alone. A survey of programming languages showed that at least 450 general-purpose programming languages and incompatible dialects were used in DoD embedded computer applications. An evaluation of existing languages occurred in 1976 against an emerging set of requirements. These evaluations resulted in four main conclusions (Whitaker, 1978):

- (1) No current language was suitable.
- (2) A single language was a desirable goal.
- (3) The state-of-the-art of language design could meet the requirements.
- (4) Development should start from a suitable language base; those recommended were Pascal, PL/I and Algol 68.

The result was the birth of a new language in 1983 called Ada. In 1995, the language was updated to reflect 10 years of use and modern advances in programming language design. The same occurred in 2005 when a number of key features were added to Ada.

Other older languages of note include PEARL, used extensively in Germany for process control applications, Mesa (Xerox Corporation, 1985), used by Xerox in their office automation equipment, and CHILL (CCITT, 1980) which was developed in response to CCITT requirements for programming telecommunication applications.

With the advent of the Internet, the Java programming language has become popular. Although initially not suitable for real-time programming, recently much effort has been dedicated to producing real-time versions of Java – this will be discussed at length later in the book.

Summary

In this chapter, a real-time system has been defined as:

any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified delay.

Two main classes of such systems have been identified: hard real-time systems, where it is absolutely imperative that responses occur within the specified

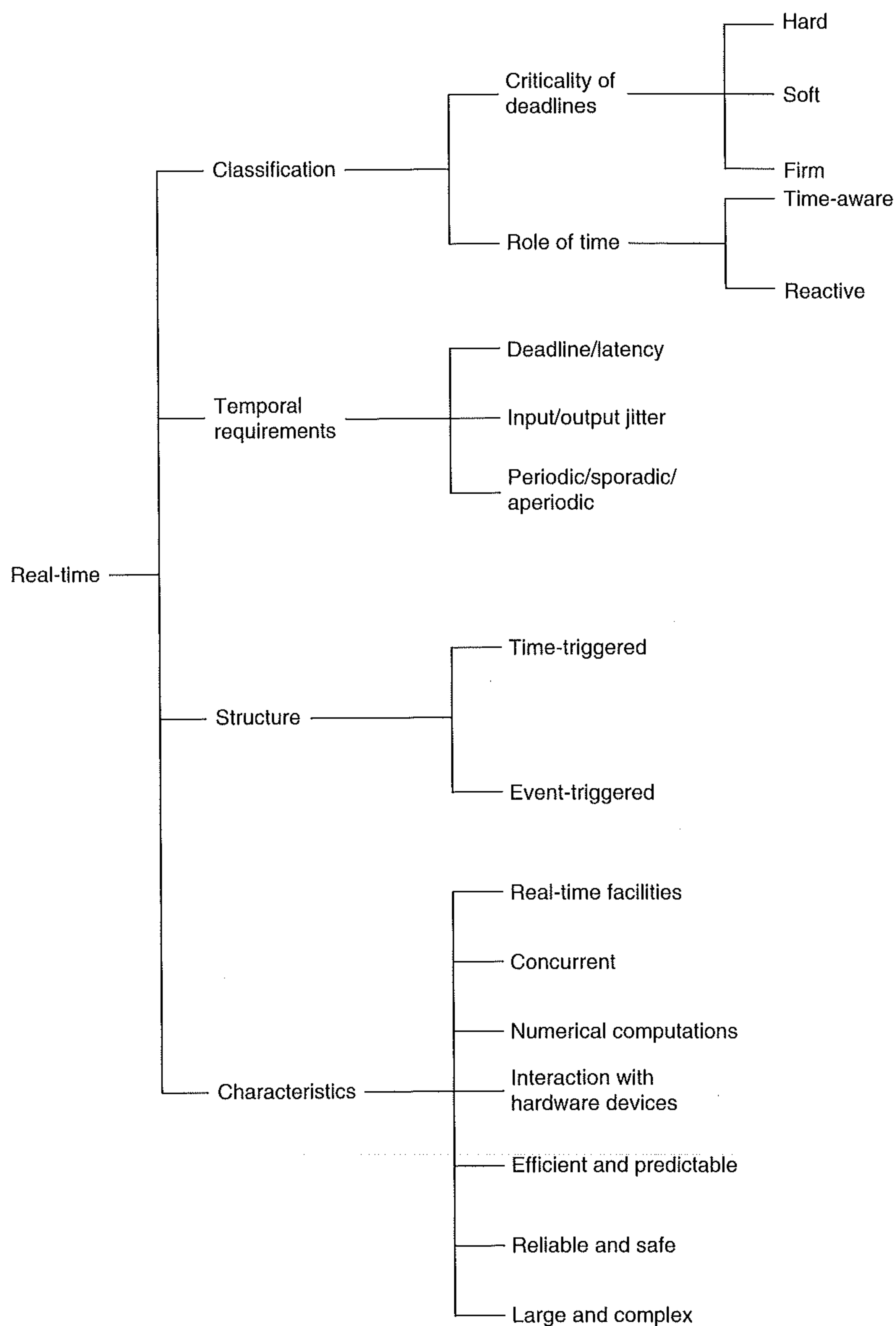


Figure 1.8 Aspects of real-time systems.

deadline; and soft real-time systems, where response times are important, but the system will still function correctly if deadlines are occasionally missed. Various types of real-time systems have been introduced including reactive systems and time-aware systems, also time-triggered and event-triggered systems.

The basic characteristics of a general real-time or embedded computer system have been considered. They are:

- real-time control;
- concurrent control of separate system components;
- low-level programming;
- support for numerical computation;
- largeness and complexity;
- extreme reliability and safety.

The main aspects associated with the term 'real-time' that have been introduced in this chapter are illustrated in Figure 1.8 opposite.

This chapter has also outlined the major stages involved in the design and implementation of real-time systems. These include requirements specification, systems design, detailed design, coding and testing. The high reliability requirements of real-time systems dictate that, wherever possible, rigorous methods should be employed.

Implementation, which is the primary focus of attention in this book, necessitates the use of a programming language. Early real-time languages lacked the expressive power to deal adequately with this application domain. More recent languages have attempted to incorporate concurrency and error-handling facilities. A discussion of these features is contained in subsequent chapters. The following general criteria were considered a useful basis for a real-time language design: security, readability, flexibility, simplicity, portability and efficiency.

Further reading

- Bailey, D. L. and Buhr, R. J. A. (1998) *Introduction to Real-Time Systems: From Design to Networking with C/C++*. Upper Saddle River, NJ: Prentice Hall.
- Booch, G., Rumbaugh, J. and Jacobson, I. (1999) *The Unified Modeling Language User Guide*. Harlow: Addison-Wesley.
- Burns, A. and Wellings, A. J. (1995) *Hard Real-Time HOOD: A Structured Design Method for Hard Real-Time Ada Systems*. New York: Elsevier.
- Cooling, J. E. (1995) *Software Design for Real-Time Systems*. London: International Thompson Computer Press.
- Douglass, B. P. (1999) *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*. Harlow: Addison-Wesley.
- Gomaa, H. (2000) *Designing Concurrent, Distributed and Real-Time Applications in UML*. Reading, MA: Addison-Wesley.
- Jacobson, I., Booch, G. and Rumbaugh, J. (1999) *The Unified Software Development Process*. Harlow: Addison Wesley Longman.
- Koptez, H. (1997) *Real-time Systems*. New York: Kluwer Academic.

- Laplante, P. (1997) *Design and Application of Real-Time Systems*. New York: Institute of Electrical & Electronic Engineers.
- Liu, J. W. S. (2000) *Real-Time Systems*. New York: Prentice Hall.
- Schneider, S. (1998) *Concurrent and Real-time Systems*. New York: Wiley.
- Schneider, S. (1999) *Concurrent and Real-time Systems: The CSP Approach*. New York: Wiley.

Exercises

- 1.1 To what extent should the choice of a design method for real-time systems be influenced by:
- (a) likely implementation language
 - (b) support tools
 - (c) reliability requirements of the application
 - (d) training requirements of staff
 - (e) marketing considerations
 - (f) previous experiences
 - (g) cost?
- 1.2 In addition to the criteria given in this chapter, what other factors could be used in assessing programming languages?
- 1.3 At what stage in the design process should the views of the end-user be obtained?
- 1.4 Should software engineers be liable for the consequences of faulty real-time systems?
- 1.5 New medicines cannot be introduced until appropriate tests and trials have been carried out. Should real-time systems be subject to similar legislation? If a proposed application is too complicated to simulate, should it be constructed?
- 1.6 Should the Ada language be the only language used in the implementation of embedded real-time systems?
- 1.7 To what extent does UML allow hard real-time systems to be designed and analysed?

Chapter 2

Reliability and fault tolerance

2.1	Reliability, failure and faults	2.8	Dynamic redundancy and exceptions
2.2	Failure modes	2.9	Measuring and predicting the reliability of software
2.3	Fault prevention and fault tolerance	2.10	Safety, reliability and dependability
2.4	<i>N</i> -version programming		Summary
2.5	Software dynamic redundancy		Further reading
2.6	The recovery block approach to software fault tolerance		Exercises
2.7	A comparison between <i>N</i> -version programming and recovery blocks		

Reliability and safety requirements are usually much more stringent for real-time and embedded systems than for other computer systems. For example, if an application which computes the solution to some scientific problem fails then it may be reasonable to abort the program, as only computer time has been lost. However, in the case of an embedded system, this may not be an acceptable action. A process control computer, for instance, responsible for the operation of a large gas furnace, cannot afford to close down the furnace as soon as a fault occurs. Instead, it must try to provide a degraded service and prevent a costly shutdown operation. More importantly, real-time computer systems may endanger human lives if they abandon control of their application. An embedded computer controlling a nuclear reactor must not let the reactor run out of control, as this may result in a core meltdown and an emission of radiation. A military avionics system should at least allow the pilot to eject before permitting the plane to crash!

It is now widely accepted that the society in which we live is totally dependent on the use of computer-based systems to support its vital functions. It is, therefore, imperative that these systems do not fail. Without wishing to define precisely what is meant by a system failure or a fault (at the moment), there are, in general, four sources of faults which can result in an embedded system failure.

- (1) Inadequate specification. It has been suggested that the great majority of software faults stem from inadequate specification (Leveson, 1986).

Included in this category are those faults that stem from misunderstanding the interactions between the program and the environment.

- (2) Faults introduced from design errors in software components.
- (3) Faults introduced by failure of one or more hardware components of the embedded system (including processors).
- (4) Faults introduced by transient or permanent interference in the supporting communication subsystem.

It is these last three types of fault which impinge on the programming language used in the implementation of an embedded system. The errors introduced by design faults are, in general, unanticipated (in terms of their consequences), whereas those from processor and network failure are, in some senses, predictable. One of the main requirements, therefore, for any real-time programming language, is that it must facilitate the construction of highly dependable systems. In this chapter, some of the general design techniques that can be used to improve the overall reliability of embedded computer systems are considered. Chapter 3 will show how **exception-handling** facilities can be used to help implement some of these design philosophies, particularly those based on **fault tolerance**.

2.1 Reliability, failure and faults

Before proceeding, more precise definitions of reliability, failures and faults are necessary. Randell et al. (1978) define the **reliability** of a system to be:

a measure of the success with which the system conforms to some authoritative specification of its behaviour.

Ideally, this specification should be complete, consistent, comprehensible and unambiguous. It should also be noted that the *response times* of the system are an important part of the specification, although discussion of the meeting of deadlines will be postponed until Chapter 11. The above definition of reliability can now be used to define a system **failure**. Again, quoting from Randell et al.:

When the behaviour of a system deviates from that which is specified for it, this is called a failure.

Section 2.9 will deal with the metrics of reliability; for the time being, *highly reliable* will be considered synonymous with a *low failure rate*.

The alert reader will have noticed that our definitions, so far, have been concerned with the *behaviour* of a system; that is, its *external* appearance. Failures result from unexpected problems internal to the system which eventually manifest themselves in the system's external behaviour. These problems are called **errors** and their mechanical or algorithmic causes are termed **faults**. A faulty component of a system is, therefore, a component which, under a particular set of circumstances during the lifetime of the system, will result in an error. Viewed in terms of state transitions, a system can be considered as a number of *external* and *internal* states. An external state which is not

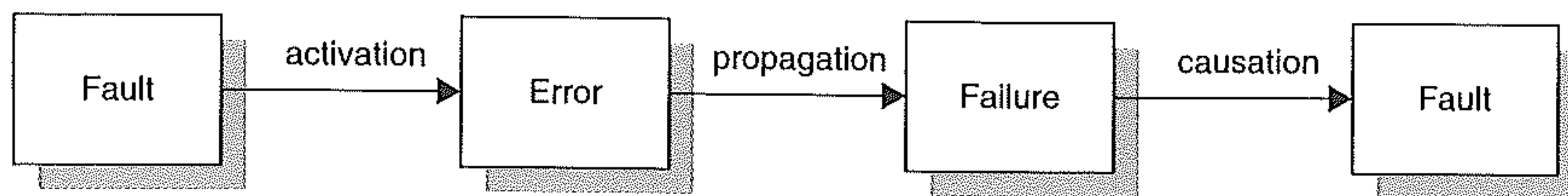


Figure 2.1 Fault, error, failure, fault chain.

specified in the behaviour of the system is regarded as a failure of the system. The system itself consists of a number of components, each with their own states, all of which contribute to the system's external behaviour. The combined states of these components are termed the internal state of the system. An internal state which is not specified is called an error and the component which produced the illegal state transition is said to be faulty.

A fault is **active** when it produces an error, and until this point it is **dormant**. Once produced, the error can be transformed into other errors via the computational process as it propagates through the system. Eventually, the error manifests itself at the boundaries of the system causing a service delivery to fail (Avizienis et al., 2004).

Of course, a system is usually composed of components; each of these may be considered as a system in its own right. Hence a failure in one system will lead to a fault in another which will result in an error and potential failure of that system. This in turn will introduce a fault into any surrounding system and so on (as illustrated in Figure 2.1).

There are many different classifications of fault types depending on the aspect of interest. For example, whether they are created during development or during operations, whether they are intentionally or accidentally created, whether they are hardware or software in origin, etc. From a real-time perspective, the duration of the fault is one of the most important aspects. Three types of fault can be distinguished.

- (1) **Transient faults** – a transient fault occurs at a particular time, remains in the system for some period and then disappears. It will initially be dormant but can become active at any time. Examples of such faults occur in hardware components which have an adverse reaction to some external interference, such as electrical fields or radioactivity. After the disturbance disappears so does the fault (although not necessarily the induced error). Many faults in communication systems are transient.
- (2) **Permanent faults** – permanent faults start at a particular time and remain in the system until they are repaired; for example, a broken wire or a software design error.
- (3) **Intermittent faults** – transient faults that occur from time to time. An example is a hardware component that is heat sensitive: it works for a time, stops working, cools down and then starts to work again.

Software faults are usually called **bugs** and it can be notoriously difficult to isolate and identify them. Over the years, particular types of bugs have been given names in an informal classification. Originally two types of software bugs were identified (Gray, 1986).¹

¹The names come from analogies with physics. The assertion that most production software bugs are ephemeral – Heisenbugs that go away when you look at them – is well known to systems programmers. Bohrbugs, like the Bohr atom, are solid, easily detected by standard techniques.

- **Bohrbugs** – these bugs are reproducible and usually identifiable. Hence they can easily be removed during testing. If they cannot be removed, then design diversity techniques can be employed during operation (see Section 2.4).
- **Heisenbugs** – these are software bugs that only activate under certain rare circumstances. A good example is code shared between concurrent tasks that is not properly synchronized. Only when two tasks happen to execute the code concurrently will the fault activate and even then the error may propagate a long way from its source before it is detected. Because of this, they often disappear when investigated – hence their name.

A particular type of Heisenbug is one that results from ‘software aging’ (Parnas, 1994). In one sense, software can be thought of as not deteriorating with age (unlike hardware). Whilst this is true, faults can remain dormant for a long time, and only become active after significant continual use of the software. These faults are normally related to resources: for example in a dynamic application where memory is constantly allocated and freed, a fault that doesn’t free unused memory will result in a **memory leak**. If this is small, the program may run for a significant period of time before memory becomes exhausted.

A good example of the effects of software ageing can be found with the use of the US Patriot missile defence system in the Gulf War in 1991 (see GAO/IMTEC-92-26 Patriot Missile Software Problem at <http://www.fas.org/spp/starwars/gao/im92026.htm>). The Patriot system was originally designed for mobile operations in Europe. The design assumed that it would only operate for a few hours at one location. During the Gulf War it was used continuously for many hours. Its main battery could last for 100 hours. After the Patriot’s radar detects an airborne object that has the characteristics of a Scud missile, the range gate (an electronic detection device within the radar system) calculates an area in the air space where the system should next look for the detected missile. The range gate filters out information about airborne objects outside its calculated area and only processes the information needed for tracking, targeting and intercepting Scuds. Finding an object within the calculated range gate area confirms that it is a Scud missile. In February 1991, a Patriot missile defence system failed to track and intercept an incoming Scud. This Scud subsequently hit an Army barracks, killing 28 people.

The reason for the failure of the Patriot’s systems is explained by considering the range gate’s prediction software, which used the Scud’s velocity and the time of the last radar detection. Time is kept continuously by the system’s internal clock in tenths of seconds held as an integer variable. The longer the system has been running, the larger the number representing time. To predict where the Scud will next appear, both time and velocity must be expressed as real numbers. The registers in the Patriot computer are only 24 bits long, and the conversion of time results in a loss of precision causing a less accurate time calculation. The effect of this inaccuracy on the range gate’s calculation is directly proportional to the target’s velocity and the length of time the system has been running. Consequently, performing the conversion after the Patriot has been running continuously for extended periods causes the range gate to shift away from the centre of the target, making it less likely that the target missile will be successfully intercepted. Table 2.1 shows the effect of this inaccuracy. After 20 hours, the target becomes outside the range gate. As with all software ageing problems, restarting the system (in this case before 20 hours of continual operational time) would clear the problem.

Hours	Seconds	Calculated time (seconds)	Inaccuracy (seconds)	Approximate shift in range gate (meters)
0	0	0	0	0
1	3600	3599.9966	.0034	7
8	28800	28799.9725	.0025	55
20	72000	71999.9313	.0687	137
48	172800	172799.8352	.1648	330
72	259200	259199.7528	.2472	494
100	360000	359999.6667	.3433	687

Table 2.1 Effect of extended run-time on Patriot operation (taken from <http://www.fas.org/spp.starwars/gao/im92026.htm>).

To create reliable systems, all types of fault must be prevented from causing erroneous system behaviour (that is failure). The difficulty this presents is compounded by the indirect use of computers in the *construction* of safety-critical systems. For example, in 1979 an error was discovered in a program used to design nuclear reactors and their supporting cooling systems. The fault that this caused in the reactor design had not been found during installation tests as it concerned the strength and structural support of pipes and valves. The program had supposedly guaranteed the attainment of earthquake safety standards in operating reactors. The discovery of the bug led to the shutting down of five nuclear power plants (Leveson, 1986).

2.2 Failure modes

A system can fail in many different ways. A designer who is using system X to implement another system, Y, usually makes some assumptions about X’s expected failure modes. If X fails differently from that which was expected then system Y may fail as a result.

A system provides services. It is, therefore, possible to classify a system’s failure modes according to the impact they have on the services it delivers. Two general domains of failure modes can be identified:

- **value failure** – the value associated with the service is in error;
- **time failure** – the service is delivered at the wrong time.

Combinations of value and timing failures are often termed **arbitrary**.

In general, a value error might still be within the correct range of values or be outside the range expected from the service. The latter is equivalent to a typing error in programming languages and is called a **constraint error**. It is usually easy to recognize this type of failure but its consequence can still be devastating. (Witness the cause of the Ariane 5 disaster where an exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer – see ‘ARIANE 5, Flight 501 Failure, Report by the Inquiry Board’ at http://klabs.org/richcontent/Reports/Failure_Reports/ariane/ariane501.htm.)

Failures in the time domain can result in the service being delivered:

- too early – the service is delivered earlier than required;
- too late – the service is delivered later than required (often called a **performance error**);
- infinitely late – the service is never delivered (often called an **omission failure**).

One further failure mode should be identified, which is where a service is delivered that is not expected. This is often called a **commission** or **impromptu** failure. It is, of course, often difficult to distinguish a failure in both the value and the time domain from a commission failure followed by an omission failure. Figure 2.2 illustrates the failure mode classification.

Given the above classification of failure modes, it is now possible to make some assumptions about how a system might fail.

- **Fail uncontrolled** – a system which can produce arbitrary errors in both the value and the time domains (including impromptu errors).
- **Fail late** – a system which produces correct services in the value domain but may suffer from a ‘late’ timing error.
- **Fail silent** – a system which produces correct services in both the value and time domains until it fails; the only failure possible is an omission failure and when this occurs all following services will also suffer an omission failure.
- **Fail stop** – a system which has all the properties of fail silent, but also permits other systems to detect that it has entered the fail-silent state.
- **Fail controlled** – a system which fails in a specified controlled manner.
- **Fail never** – a system which always produces correct services in both the value and the time domain.

Other assumptions and classifications are clearly possible, but the above list will suffice for this book.

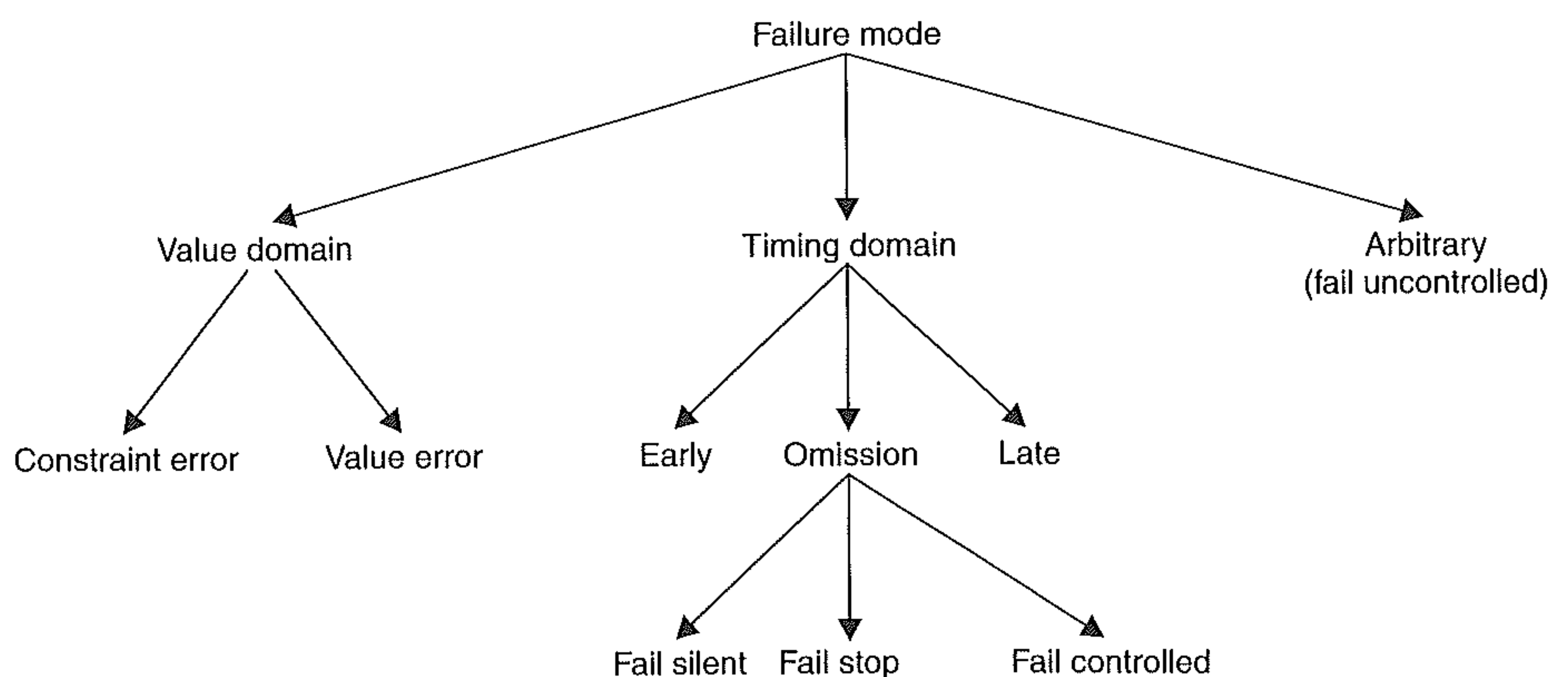


Figure 2.2 Failure mode classification.

2.3 Fault prevention and fault tolerance

Two approaches that can help designers improve the reliability of their systems can be distinguished (Anderson and Lee, 1990). The first is known as **fault prevention**; this attempts to eliminate any possibility of faults creeping into a system before it goes operational. The second is **fault tolerance**; this enables a system to continue functioning even in the presence of faults. Both approaches attempt to produce systems which have well-defined failure modes.

2.3.1 Fault prevention

There are two stages to fault prevention: **fault avoidance** and **fault removal**.

Fault avoidance attempts to limit the introduction of potentially faulty components during the construction of the system. For hardware this may entail (Randell et al., 1978):

- the use of the most reliable components within the given cost and performance constraints;
- the use of thoroughly-refined techniques for the interconnection of components and the assembly of subsystems;
- packaging the hardware to screen out expected forms of interference.

The software components of large embedded systems are nowadays much more complex than their hardware counterparts. It is virtually impossible in all cases to write fault-free programs. However the quality of software can be improved by:

- rigorous, if not formal, specification of requirements (for example, B or Z);
- the use of proven design methodologies (for example, those based on UML, such as Real-Time UML (Douglass, 1999));
- the use of analysis tools to verify key program properties (such as model checkers or proof checkers to ensure multitask programs are free from deadlock);
- the use of languages with facilities for data abstraction and modularity (for example, Ada or Java);
- the use of software engineering tools to help manipulate software components and thereby manage complexity (for example, configuration management tools such as CVS).

In spite of fault avoidance techniques, faults will inevitably be present in the system after its construction. In particular, there may be design errors in both hardware and software components. The second stage of fault prevention, therefore, is *fault removal*. This normally consists of procedures for finding and then removing the causes of errors. Although techniques such as design reviews, program verification and code inspections may be used, emphasis is usually placed on system testing. Unfortunately, system testing can never be exhaustive and remove all potential faults. In particular, the following problems exist.

- A test can only be used to show the presence of faults, not their absence.

- It is sometimes impossible to test under realistic conditions – one of the major causes for concern over the American Strategic Defense Initiative (SDI)² was the impossibility of testing any system realistically except under battle conditions. Most tests are done with the system in simulation mode and it is difficult to guarantee that the simulation is accurate. The last French nuclear testing at Mururoo in the Pacific during 1995 was allegedly to allow data to be collected so that future tests would not be necessary but could be simulated accurately.
- Errors that have been introduced at the requirements stage of the system's development may not manifest themselves until the system goes operational. For example, in the design of the F18 aircraft an erroneous assumption was made concerning the length of time taken to release a wing-mounted missile. The problem was discovered only during operation when the missile failed to separate from the launcher after ignition, causing the aircraft to go violently out of control (Leveson, 1986).

In spite of all the testing and verification techniques, hardware components will fail; the fault prevention approach will, therefore, be unsuccessful when either the frequency or duration of repair times are unacceptable, or the system is inaccessible for maintenance and repair activities. An extreme example of the latter is the crewless spacecraft Voyager.

2.3.2 Fault tolerance

Because of the inevitable limitations of the fault prevention approach, designers of embedded systems must consider the use of fault tolerance. Of course, this does not mean that attempts at preventing faulty systems from becoming operational should be abandoned. However, this book will focus on fault tolerance rather than fault prevention.

Several different levels of fault tolerance can be provided by a system.

- **Full fault tolerance** – the system continues to operate in the presence of faults, albeit for a limited period, with no significant loss of functionality or performance.
- **Graceful degradation** (or fail-soft) – the system continues to operate in the presence of errors, accepting a partial degradation of functionality or performance during recovery or repair.
- **Fail safe** – the system maintains its integrity while accepting a temporary halt in its operation.

The level of fault tolerance required will depend on the application. Although in theory most safety-critical systems require full fault tolerance, in practice many settle for graceful degradation. In particular, those systems which can suffer physical damage, such as combat aircraft, may provide several degrees of graceful degradation. Also, with highly complex applications which have to operate on a continuous basis (they have *high availability* requirements) graceful degradation is a necessity, as full fault tolerance is

²This was proposed by President Reagan in the 1980s. Its goal was to use ground-based and space-based systems to protect the US from attacks by ballistic missiles. It was never fully developed or deployed.

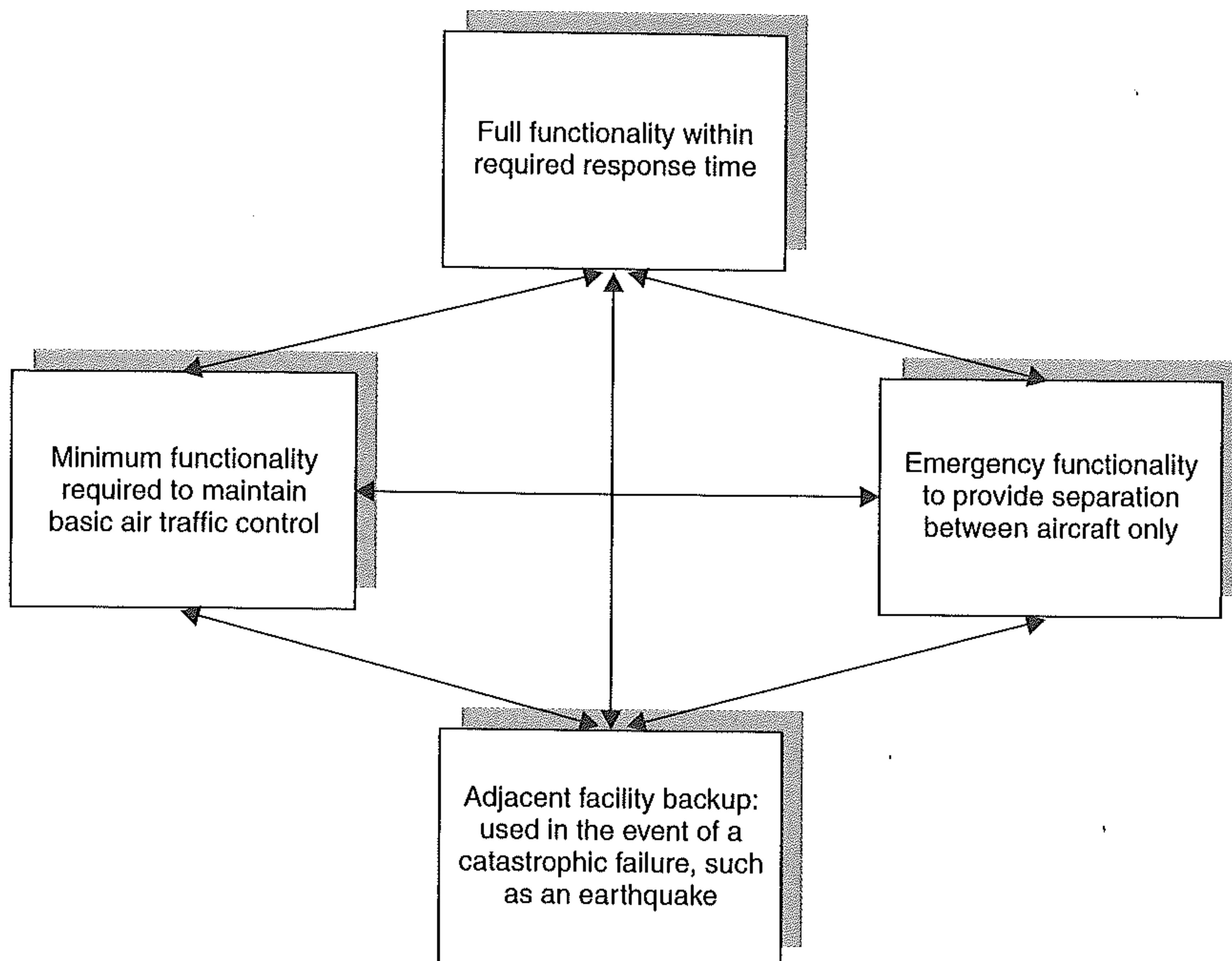


Figure 2.3 Graceful degradation and recovery in an air traffic control system.

not achievable for indefinite periods. For example, the Federal Aviation Administration’s Advanced Automation System, which provides automated services to both *en route* and terminal air traffic controllers throughout the USA, has three levels of graceful degradation for its area control computer couplers (Avizienis and Ball, 1987). This is illustrated in Figure 2.3.

In some situations, it may simply be necessary to shut down the system in a safe state. These fail-safe systems attempt to limit the amount of damage caused by a failure. For example, the A310 Airbus’s slat and flap control computers, on detecting an error on landing, restore the system to a safe state and then shut down. In this situation, a safe state is having both wings with the same settings; only asymmetric settings are hazardous in landing (Martin, 1982).

Early approaches to the design of fault-tolerant systems made three assumptions.

- (1) The algorithms of the system have been correctly designed.
- (2) All possible failure modes of the components are known.
- (3) All possible interactions between the system and the environment have been foreseen.

However, the increasing complexity of computer software and the introduction of multi-core hardware components mean that it is no longer possible to make these assumptions (if it ever was). Consequently, both anticipated and unanticipated faults must be catered for. The latter include both hardware and software design faults.

2.3.3 Redundancy

All techniques for achieving fault tolerance rely on extra elements introduced into the system to detect and recover from faults. These components are redundant in the sense that they are not required for the system's normal mode of operation. This is often called **protective redundancy**. The aim of fault tolerance is to minimize redundancy while maximizing the reliability provided, subject to the cost, size and power constraints of the system. Care must be taken in structuring fault-tolerant systems because the added components inevitably increase the complexity of the overall system. This itself can lead to *less* reliable systems. For example, the first launch of the Space Shuttle was aborted because of a synchronization difficulty with the replicated computer systems (Garman, 1981). To help reduce problems associated with the interaction between redundant components, it is therefore advisable to separate out the fault-tolerant components from the rest of the system.

There are several different classifications of redundancy, depending on which system components are under consideration and which terminology is being used. Software fault tolerance is the main focus of this chapter and therefore only passing reference will be made to hardware redundancy techniques. For hardware, Anderson and Lee (1990) distinguish between **static** (or masking) and **dynamic** redundancy. With static redundancy, redundant components are used inside a system (or subsystem) to hide the effects of faults. An example of static redundancy is **Triple Modular Redundancy** (TMR). TMR consists of three identical subcomponents and majority voting circuits. The circuits compare the output of all the components, and if one differs from the other two that output is masked out. The assumption here is that the fault is not due to a common aspect of the subcomponents (such as a design error), but is either transient or due to component deterioration. Clearly, to mask faults from more than one component requires more redundancy. The general term **N Modular Redundancy** (NMR) is therefore used to characterize this approach.

Dynamic redundancy is the redundancy supplied inside a component which indicates explicitly or implicitly that the output is in error. It therefore provides an **error detection** facility rather than an error-masking facility; recovery must be provided by another component. Examples of dynamic redundancy are checksums on communication transmissions and parity bits on memories.

For fault tolerance of software design errors, two general approaches can be identified. The first is analogous to hardware masking redundancy and is called *N-version programming*. The second is based on error detection and recovery; it is analogous to dynamic redundancy in the sense that the recovery procedures are brought into action only after an error has been detected.

2.4 N-version programming

The success of hardware TMR and NMR has motivated a similar approach to software fault tolerance. Here, the approach is used to focus on detecting design faults. In fact, this approach (which is now known as *N-version programming*) was first advocated by Babbage in 1837 (Randell, 1982):

When the formula is very complicated, it may be algebraically arranged for computation in two or more distinct ways, and two or more sets of cards

may be made. If the same constants are now employed with each set, and if under these circumstances the results agree, we may then be quite secure of the accuracy of them all.

N-version programming is defined as the independent generation of *N* (where *N* is greater than or equal to 2) functionally equivalent programs from the same initial specification (Chen and Avizienis, 1978). The independent generation of *N* programs means that *N* individuals or groups produce the required *N* versions of the software *without interaction* (for this reason *N*-version programming is often called **design diversity**). Once designed and written, the programs execute concurrently with the same inputs and their results are compared by a **driver process**. In principle, the results should be identical, but in practice there may be some difference, in which case the consensus result, assuming there is one, is taken to be correct.

N-version programming is based on the assumptions that a program can be completely, consistently and unambiguously specified, and that programs which have been developed independently will fail independently. That is, there is no relationship between the faults in one version and the faults in another. This assumption may be invalidated if each version is written in the same programming language, because errors associated with the implementation of the language may be common between versions. Consequently, different programming languages and different development environments should be used. Alternatively, if the same language is used, compilers and support environments from different manufacturers should be employed. Furthermore, in either case, to protect against physical faults, the *N* versions must be distributed to separate machines which have fault-tolerant communication lines. On the Boeing 777 flight control system, a single Ada program was produced but three different processors and three distinct compilers were used to obtain diversity.

The *N*-version program is controlled by a driver process which is responsible for:

- invoking each of the versions;
- waiting for the versions to complete;
- comparing and acting on the results.

So far it has been implicitly assumed that the programs or processes run to completion before the results are compared, but for embedded systems this often will not be the case; such processes may never complete. The driver and *N* versions must, therefore, communicate during the course of their executions.

It follows that these versions, although independent, must interact with the driver program. This interaction is specified in the requirements for the versions. It consists of three components (Chen and Avizienis, 1978):

- (1) comparison vectors;
- (2) comparison status indicators;
- (3) comparison points.

How the versions communicate and synchronize with the driver will depend on the programming language used and its model of concurrency (see Chapters 4, 5 and 6). If different languages are used for different versions, then a real-time operating system

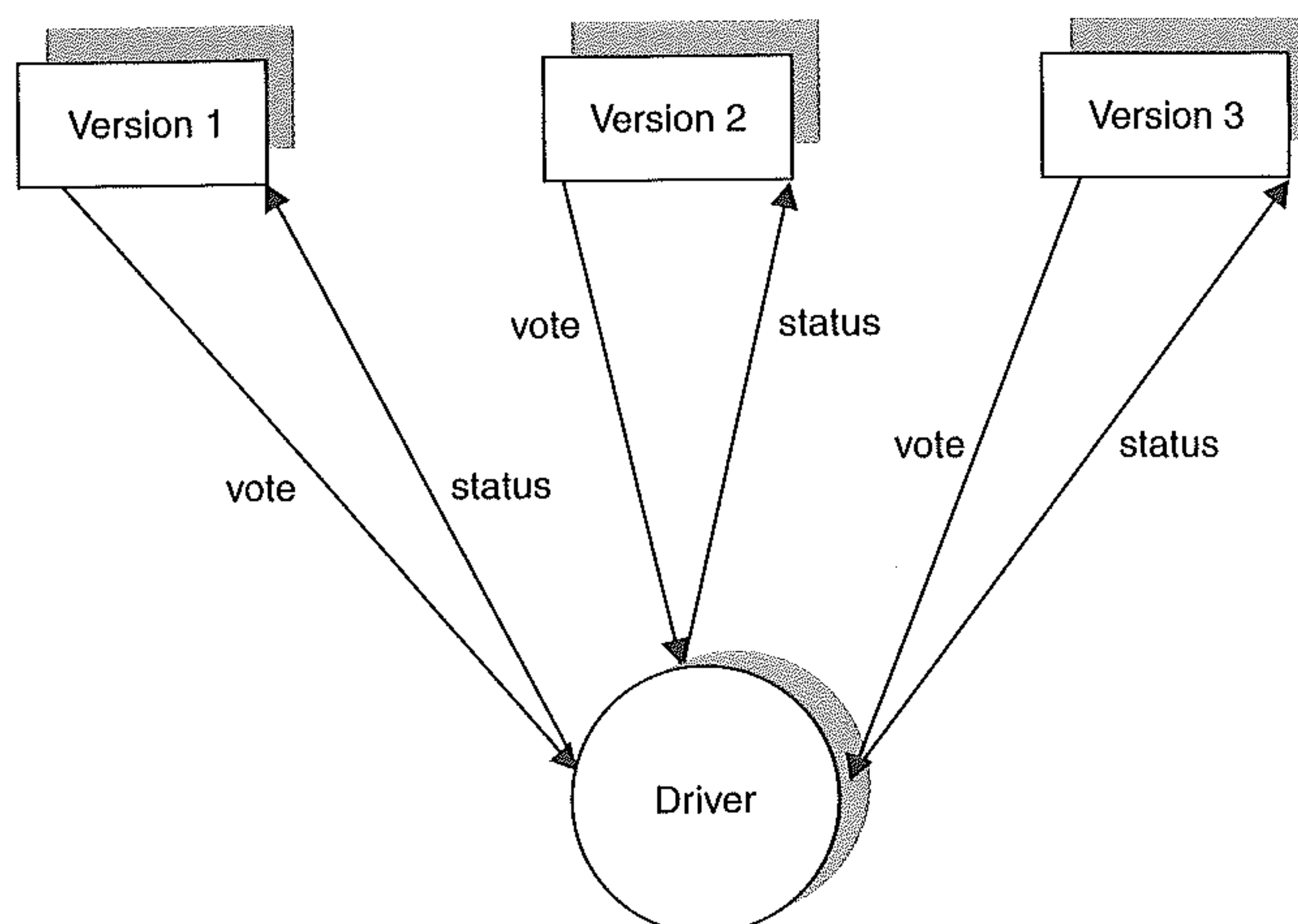


Figure 2.4 *N*-version programming.

will usually provide the means of communication and synchronization. The relationship between the N versions and the driver for an $N = 3$ version system is shown diagrammatically in Figure 2.4.

Comparison vectors are the data structures which represent the outputs, or votes, produced by the versions plus any attributes associated with their calculation; these must be compared by the driver. For example, in an air traffic control system, if the values being compared are the positions of aircraft, an attribute may indicate whether the values were the result of a recent radar reading or calculated on the basis of old readings.

The comparison status indicators are communicated from the driver to the versions; they indicate the actions that each version must perform as a result of the driver's comparison. Such actions will depend on the outcome of the comparison: whether the votes agreed and whether they were delivered on time. Possible outcomes include:

- continuation;
- termination of one or more versions;
- continuation after changing one or more votes to the majority value.

The comparison points are the points in the versions where they must communicate their votes to the driver process. As Hecht and Hecht (1986) point out, an important design decision is the frequency with which the comparisons are made. This is the **granularity** of the fault tolerance provision. Fault tolerance of large granularity, that is, infrequent comparisons, will minimize the performance penalties inherent in the comparison strategies and permit a large measure of independence in the version design. However, a large granularity will probably produce a wide divergence in the results obtained because of the greater number of steps carried out between comparisons. The problems of vote comparison or voting (as it is often called) are considered in the next subsection. Fault tolerance of a fine granularity requires commonality of program structures at a detailed level, and therefore reduces the degree of independence between

versions. A frequent number of comparisons also increase the overheads associated with this technique.

2.4.1 Vote comparison

Crucial to N -version programming is the efficiency and the ease with which the driver program can compare votes and decide whether there is any disagreement. For applications which manipulate text or perform integer arithmetic there will normally be a single correct result; the driver can easily compare votes from different versions and choose the majority decision.

Unfortunately, not all results are of an exact nature. In particular, where votes require the calculation of real numbers, it will be unlikely that different versions will produce exactly the same result. This might be due to the inexact hardware representation of real numbers or the data sensitivity of a particular algorithm. The techniques used for comparing these types of results are called **inexact voting**. One simple technique is to conduct a range check using a previous estimation or a median value taken from all N results. However, it can be difficult to find a general inexact voting approach.

Another difficulty associated with finite-precision arithmetic is the so-called **consistent comparison problem** (Brilliant et al., 1987). The trouble occurs when an application has to perform a comparison based on a finite value given in the specification; the result of the comparison then determines the course of action to be taken. As an example, consider a process control system which monitors temperature and pressure sensors and then takes appropriate actions according to their values to ensure the integrity of the system. Suppose that when either of these readings passes a threshold value some corrective course of action must be taken. Now consider a 3-version software system (V_1, V_2, V_3) each of which must read both sensors, decide on some action and then vote on the outcome (there is no communication between the versions until they vote). As a result of finite-precision arithmetic, each version will calculate different values (say T_1, T_2, T_3 for the temperature sensor and P_1, P_2, P_3 for the pressure sensor). Assuming that the threshold value for temperature is T_{th} and for pressure P_{th} , the consistent comparison problem occurs when both readings are around their threshold values.

The situation might occur where T_1 and T_2 are just below T_{th} and T_3 just above; consequently V_1 and V_2 will follow their normal execution paths and V_3 will take some corrective action. Now if versions V_1 and V_2 proceed to another comparison point, this time with the pressure sensor, then it is possible that P_1 could be just below and P_2 just above P_{th} . The overall result will be that all three versions will have followed different execution paths, and therefore produce different results, each of which is valid. This process is represented diagrammatically in Figure 2.5.

At first sight, it might seem appropriate to use inexact comparison techniques and assume that the values are equal if they differ by a tolerance Δ , but as Brilliant et al. (1987) point out, the problem reappears when the values are close to the threshold value $\pm\Delta$.

Still further problems exist with vote comparison when multiple solutions to the same problem naturally exist. For example, a quadratic equation may have more than one solution. Once again disagreement is possible, even though no fault has occurred (Anderson and Lee, 1990).

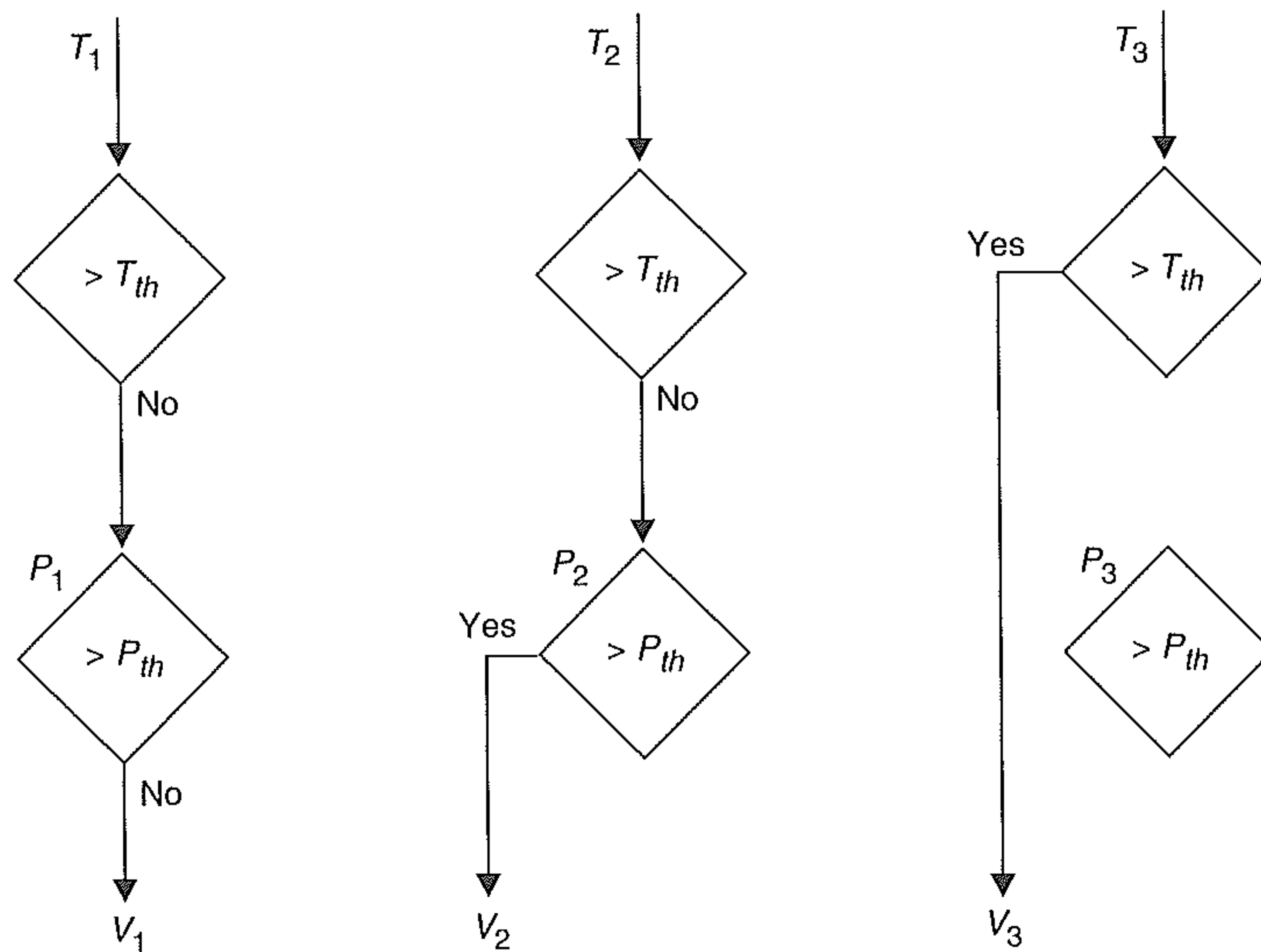


Figure 2.5 Consistent comparison problem with three versions.

2.4.2 Principal issues in *N*-version programming

It has been shown that the success of *N*-version programming depends on several issues, which are now briefly reviewed.

- (1) **Initial specification** – it has been suggested that the great majority of software faults stem from inadequate specification (Leveson, 1986). Current techniques are a long way from producing complete, consistent, comprehensible and unambiguous specifications, although formal specification methods are proving a fruitful line of research. Clearly a specification error will manifest itself in all *N* versions of the implementation.
- (2) **Independence of design effort** – some experiments (Knight et al., 1985; Avizienis et al., 1988; Brilliant et al., 1990; Eckhardt et al., 1991; Hatton, 1997) have been undertaken to test the hypothesis that independently produced software will display distinct failures; however, they produce conflicting results. Knight et al. (1985) have shown that for a particular problem with a thoroughly refined specification, the hypothesis had to be rejected at the far from adequate 99% confidence level. In contrast, Avizienis et al. (1988) found that it was very rare for identical faults to be found in two versions of a six-version system. In comparing their results and those produced by Knight et al., they concluded that the problem addressed by Knight et al. had limited potential for diversity, the programming process was rather informally formulated, testing was limited, and the acceptance test was totally inadequate according to common industrial standards. Avizienis et al. claim that the rigorous application of the *N*-version programming paradigm would have led to the elimination of all of the errors reported by Knight et al. before the acceptance of the system. However, there is concern that where part of a specification is complex this will inevitably lead to a lack of understanding of the requirements by

all the independent teams. If these requirements also refer to rarely occurring input data, then common design errors may not be caught during system testing. In more recent years, studies by Hatton (1997) found that a three-version system is still around five to nine times more reliable than a single-version high-quality system.

- (3) **Adequate budget** – with most embedded systems, the predominant cost is software. A three-version system will therefore almost triple the budget requirement and cause problems for maintenance personnel. In a competitive environment, it is unlikely that a potential contractor will propose an N -version technique unless it is mandatory. Furthermore, it is unclear whether a more reliable system would be produced if the resources potentially available for constructing N versions were instead used to produce a single version.

It has also been shown that in some instances it is difficult to find inexact voting algorithms, and that unless care is taken with the consistent comparison problem, votes will differ even in the absence of faults.

Although N -version programming may have a role in producing reliable software it should be used with care and in conjunction with other techniques; for example, those discussed below.

2.5 Software dynamic redundancy

N -version programming is the software equivalent of static or masking redundancy, where faults inside a component are hidden from the outside. It is static because each version of the software has a fixed relationship with every other version and the driver; and because it operates whether or not faults have occurred. With **dynamic** redundancy, the redundant components only come into operation *when* an error has been detected.

This technique of fault tolerance has four constituent phases (Anderson and Lee, 1990).

- (1) **Error detection** – most faults will eventually manifest themselves in the form of an error; no fault tolerance scheme can be utilized until that error is detected.
- (2) **Damage confinement and assessment** – when an error has been detected, it must be decided to what extent the system has been corrupted (this is often called *error diagnosis*); the delay between a fault occurring and the manifestation of the associated error means that erroneous information could have spread throughout the system.
- (3) **Error recovery** – this is one of the most important aspects of fault tolerance. Error recovery techniques should aim to transform the corrupted system into a state from which it can continue its normal operation (perhaps with degraded functionality).
- (4) **Fault treatment and continued service** – an error is a symptom of a fault; although the damage may have been repaired, the fault may still exist, and therefore the error may recur unless some form of maintenance is undertaken.

Although these four phases of fault tolerance are discussed under software dynamic redundancy techniques, they can clearly be applied to N -version programming. As

Anderson and Lee (1990) have noted: error detection is provided by the driver which does the vote checking; damage assessment is not required because the versions are independent; error recovery involves discarding the results in error, and fault treatment is simply ignoring the version determined to have produced the erroneous value. However, if all versions have produced differing votes then error detection takes place, but there are *no* recovery facilities.

The next sections briefly cover the above phases of fault tolerance. For a fuller discussion, the reader is referred to Anderson and Lee (1990).

2.5.1 Error detection

The effectiveness of any fault-tolerant system depends on the effectiveness of its error detection techniques. Two classes of error detection techniques can be identified.

- **Environmental detection** – these are the errors which are detected in the environment in which the program executes. They include those that are detected by the hardware, such as ‘illegal instruction executed’, ‘arithmetic overflow’ and ‘protection violation’. They also include errors detected by the run-time support system for the real-time programming language; for example, ‘array bounds error’, ‘null pointer referenced’ and ‘value out of range’. These types of error will be considered in the context of the Ada and Java programming languages in Chapter 3.
- **Application detection** – these are the errors that are detected by the application itself. The majority of techniques that can be used by the application fall into the following broad categories.
 - **Replication checks** – it has been shown that N -version programming can be used to tolerate software faults and that the technique can be used to provide error detection (by using two-version redundancy).
 - **Timing checks** – two types of timing check can be identified. The first involves a **watchdog timer** process that, if not reset within a certain period by a component, assumes that the component is in error. The software component must continually reset the timer to indicate that it is functioning correctly. In embedded systems, where timely responses are important, a second type of check is required. These enable the detection of faults associated with missed deadlines. Where deadline scheduling is performed by the underlying run-time support system, the detection of missed deadlines can be considered to be part of the environment. For example, with the Real-Time Specification for Java it is the real-time JVM that detects deadline misses. However, an Ada programmer must detect such an error in the application. The issue of tolerating timing faults is covered in detail in Chapter 13. Of course, timing checks do *not* ensure that a component is functioning correctly, only that it is functioning on time! Time checks should therefore be used in conjunction with other error detection techniques.
 - **Reversal checks** – these are feasible in components where there is a one-to-one (isomorphic) relationship between the input and the output. Such a check takes the output, calculates what the input should be, and then compares the value with the actual input. For example, for a component which finds the square root of a number, the reversal check is simply to square the output

and compare it with the input. (Note that inexact comparison techniques may have to be used when dealing with real numbers.)

- **Coding checks** – coding checks are used to test for the corruption of data. They are based on redundant information contained within the data. For example, a value (checksum) may be calculated and sent with the actual data to be transmitted over a communication network. When the data is received, the value can be recalculated and compared with the checksum.
- **Reasonableness checks** – these are based on knowledge of the internal design and construction of the system. They check that the state of data or value of an object is reasonable, based on its intended use. Typically with modern real-time languages, much of the information necessary to perform these checks can be supplied by programmers, as type information associated with data objects. For example, in Ada integer objects which are constrained to be within certain values can be represented by subtypes of integers which have explicit ranges. Range violation can then be detected by the run-time support system.

Sometimes explicit reasonableness checks are included in software components; these are commonly called **assertions** and take a logical expression which evaluates at run-time to true if no error is detected.

- **Structural checks** – structural checks are used to check the integrity of data objects such as lists or queues. They might consist of counts of the number of elements in the object, redundant pointers or extra status information.
- **Dynamic reasonableness checks** – with output emitted from some digital controllers, there is usually a relationship between any two consecutive outputs. Hence an error can be assumed if a new output is too different from the previous value.

Note that many of the above techniques may be applied also at the hardware level and therefore may result in ‘environmental errors’.

2.5.2 Damage confinement and assessment

As there can be some delay between a fault occurring and an error being detected, it is necessary to assess any damage that may have occurred. While the type of error that was detected will give the error-handling routine some idea of the damage, erroneous information could have spread throughout the system and into its environment. Thus damage assessment will be closely related to the damage confinement precautions that were taken by the system’s designers. Damage confinement is concerned with structuring the system so as to minimize the damage caused by a faulty component. It is also known as **firewalling**.

There are two techniques that can be used for structuring systems which will aid damage confinement: **modular decomposition** and **atomic actions**. With modular decomposition the emphasis is simply that the system should be broken down into components where each component is represented by one or more modules. Interaction between components then occurs through well-defined interfaces, and the internal details of the modules are hidden and not directly accessible from the outside. This makes it more difficult for an error in one component to be indiscriminately passed to another.

Modular decomposition provides a *static* structure to the software system in that most of that structure is lost at run-time. Equally important to damage confinement is the *dynamic* structure of the system as it facilitates reasoning about the run-time behaviour of the software. One important dynamic structuring technique is based on the use of atomic actions.

The activity of a component is said to be atomic if there are *no* interactions between the activity and the system for the duration of the action.

That is, to the rest of the system an atomic action appears to be *indivisible* and takes place *instantaneously*. No information can be passed from within the atomic action to the rest of the system and vice versa. Atomic actions are often called **transactions** or **atomic transactions**. They are used to move the system from one consistent state to another and constrain the flow of information between components. Where two or more components share a resource then damage confinement will involve constraining access to that resource. The implementation of this aspect of atomic actions, using the communication and synchronization primitives found in modern real-time languages, will be considered in Chapter 7.

Other techniques which attempt to restrict access to resources are based on **protection mechanisms**, some of which may be supported by hardware. For example, each resource may have one or more modes of operation each with an associated access list (for example, read, write and execute). An activity of a component, or process, will also have an associated mode. Every time a process accesses a resource, the intended operation can be compared against its **access permissions** and, if necessary, access is denied.

In the time domain, damage confinement techniques focus on resource reservation techniques. Budgets can be given to processes that can be policed at run-time. This topic is covered in detail in Chapter 13.

2.5.3 Error recovery

Once an error situation has been detected and the damage assessed, error recovery procedures must be initiated. This is probably one of the most important phases of any fault-tolerance technique. It must transform an erroneous system state into one which can continue its normal operation, although perhaps with a degraded service. Two approaches to error recovery have been proposed: **forward** and **backward** recovery.

Forward error recovery attempts to continue from an erroneous state by making selective corrections to the system state. For embedded systems, this may involve making safe any aspect of the controlled environment which may be hazardous or damaged because of the failure. Although forward error recovery can be efficient, it is system specific and depends on accurate predictions of the location and cause of errors (that is, damage assessment). Examples of forward recovery techniques include redundant pointers in data structures and the use of self-correcting codes, such as Hamming Codes. An abort, or asynchronous exception, facility may also be required during the recovery action if more than one process is involved in providing the service when the error occurred.

Backward error recovery relies on restoring the system to a safe state previous to that in which the error occurred. An alternative section of the program is then executed. This has the same functionality as the fault-producing section, but uses a different algorithm. As with *N*-version programming, it is hoped that this alternative approach

will *not* result in the same fault recurring. The point to which a process is restored is called a **recovery point** and the act of establishing it is usually termed **checkpointing**. To establish a recovery point, it is necessary to save appropriate system state information at run-time.

State restoration has the advantage that the erroneous state has been cleared and that it does not rely on finding the location or cause of the fault. Backward error recovery can therefore be used to recover from unanticipated faults including design errors. However, its disadvantage is that it cannot undo any effects that the fault may have had in the environment of the embedded system; it is difficult to undo a missile launch, for example. Furthermore, backward error recovery can be time-consuming in execution, which may preclude its use in some real-time applications. For instance, operations involving sensor information may be time dependent, therefore costly state restoration techniques may simply not be feasible. Consequently, to improve performance **incremental checkpointing** approaches have been considered. The **recovery cache** is an example of such a system (Anderson and Lee, 1990). Other approaches include audit trails or logs; in these cases, the underlying support system must undo the effects of the process by reversing the actions indicated in the log.

With concurrent processes that interact with each other, state restoration is not as simple as so far portrayed. Consider two processes depicted in Figure 2.6. Process P_1 establishes recovery points R_{11} , R_{12} and R_{13} . Process P_2 establishes recovery points R_{21} and R_{22} . Also, the two processes communicate and synchronize their actions via IPC_1 , IPC_2 , IPC_3 and IPC_4 . The abbreviation *IPC* is used to indicate Inter-Process Communication.

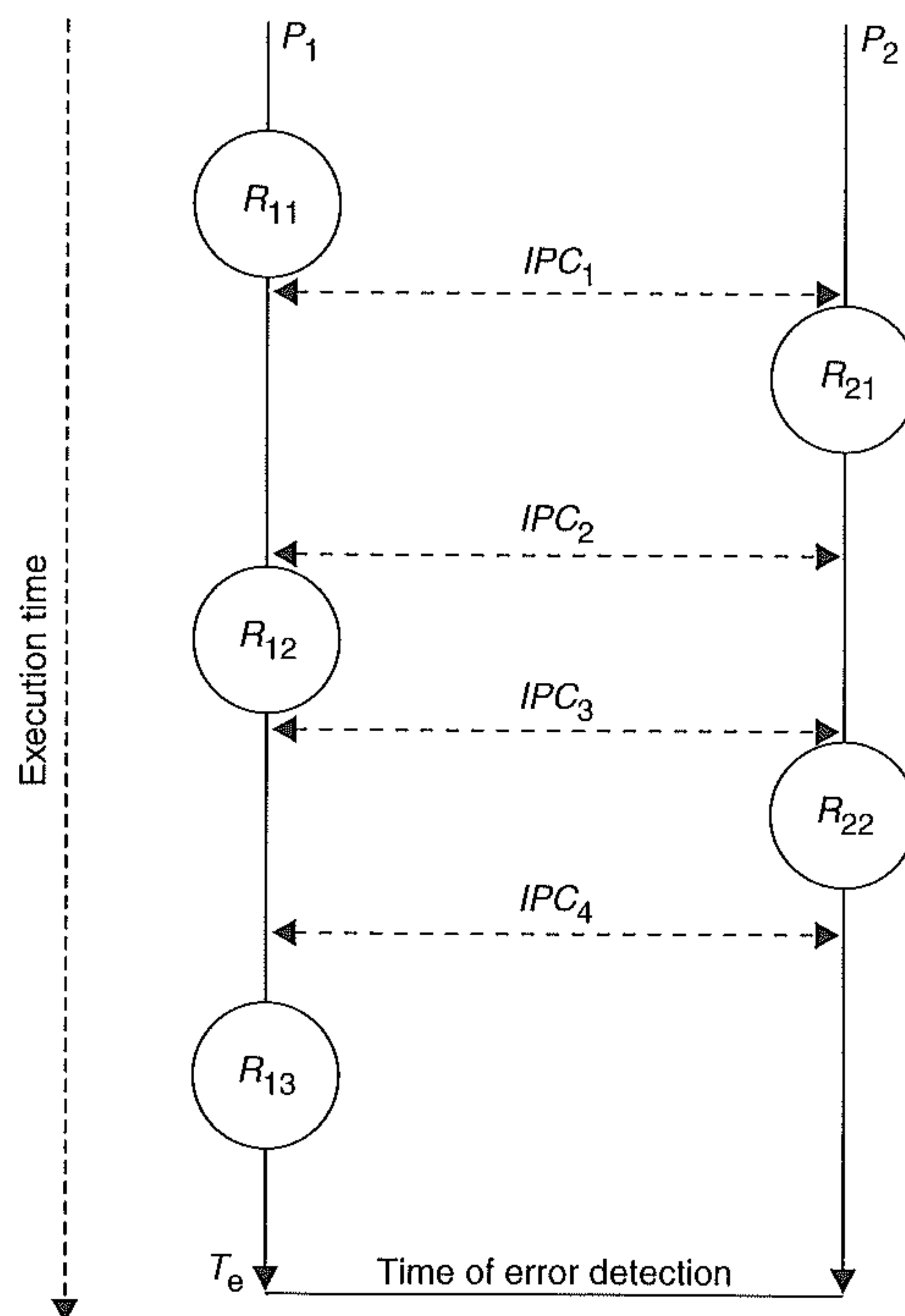


Figure 2.6 The domino effect.

If P_1 detects an error at T_e then it is simply rolled back to recovery point R_{13} . However, consider the case where P_2 detects an error at T_e . If P_2 is rolled back to R_{22} then it must undo the communication IPC_4 with P_1 ; this requires P_1 to be rolled back to R_{12} . But if this is done, P_2 must be rolled back to R_{21} to undo communication IPC_3 , and so on. The result will be that both processes will be rolled back to the beginning of their interaction with each other. In many cases, this may be equivalent to aborting both processes! This phenomenon is known as the **domino effect**.

Obviously, if the two processes do not interact with each other then there will be no domino effect. When more than two processes interact, the possibility of the effect occurring increases. In this case, consistent recovery points must be designed into the system so that an error detected in one process will not result in a total rollback of all the processes with which it interacts; instead, the processes can be restarted from a consistent set of recovery points. These **recovery lines**, as they are often called, are closely linked with the notion of atomic actions, introduced earlier in this section. The issue of error recovery in concurrent processes will be revisited in Chapter 7. For the remainder of this chapter, sequential systems only will be considered.

The concepts of forward and backward error recovery have been introduced; each has its advantages and disadvantages. Not only do embedded systems have to be able to recover from unanticipated errors but they also must be able to respond in finite time; they may therefore require *both* forward and backward error recovery techniques. The expression of backward error recovery in sequential experimental programming languages will be considered in the next section. Mechanisms for forward error recovery will not be considered further in this chapter because it is difficult to provide in an application-independent manner. However, in the next chapter the implementation of both forms of error recovery is considered within the common framework of exception handling.

2.5.4 Fault treatment and continued service

An error is a manifestation of a fault, and although the error recovery phase may have returned the system to an error-free state, the error may recur. Therefore the final phase of fault tolerance is to eradicate the fault from the system so that normal service can be continued.

The automatic treatment of faults is difficult to implement and tends to be system-specific. Consequently, some systems make no provision for fault treatment, assuming that all faults are transient; others assume that error recovery techniques are sufficiently powerful to cope with recurring faults.

Fault treatment can be divided into two stages: fault location and system repair. Error detection techniques can help to trace the fault to a component. For a hardware component this may be accurate enough and the component can simply be replaced. A software fault can be removed in a new version of the code. However, in most non-stop applications it will be necessary to modify the program while it is executing. This presents a significant technical problem, but will not be considered further here.

2.6 The recovery block approach to software fault tolerance

Recovery blocks (Horning et al., 1974) are **blocks** in the normal programming language sense except that at the entrance to the block is an automatic **recovery point** and at

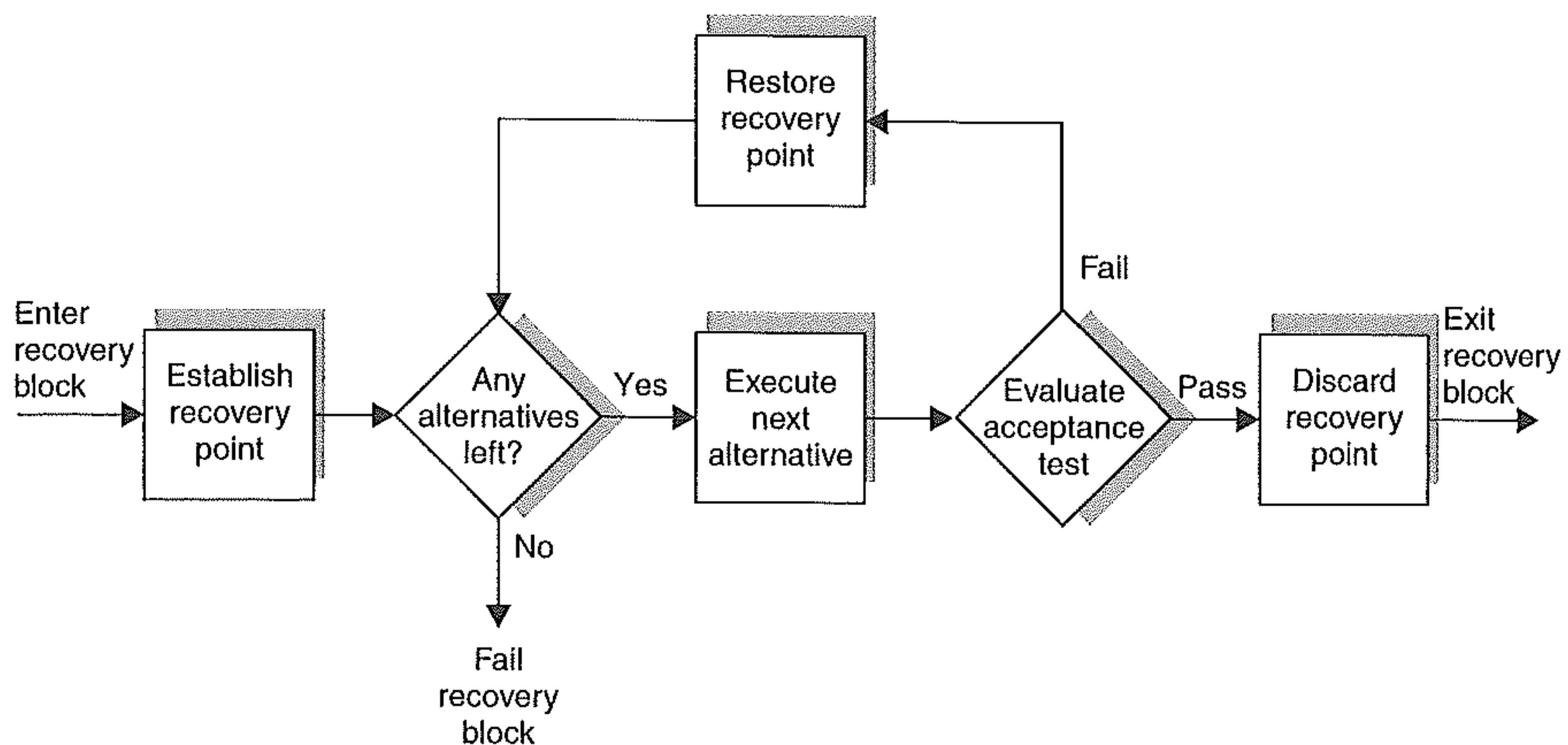


Figure 2.7 Recovery block mechanism.

the exit an **acceptance test**. The acceptance test is used to test that the system is in an acceptable state after the execution of the block (or **primary module** as it is often called). The failure of the acceptance test results in the program being restored to the recovery point at the beginning of the block and an **alternative module** being executed. If the alternative module also fails the acceptance test then again the program is restored to the recovery point and yet another module is executed, and so on. If all modules fail then the block fails and recovery must take place at a higher level. The execution of a recovery block is illustrated in Figure 2.7.

In terms of the four phases of software fault tolerance: error detection is achieved by the acceptance test, damage assessment is not needed as backward error recovery is assumed to clear all erroneous states, and fault treatment is achieved by use of a stand-by spare.

Although no commercially available real-time programming language has language features for exploiting recovery blocks, some experimental systems have been developed (Shrivastava, 1978; Purtilo and Jalote, 1991). A possible syntax for recovery blocks is illustrated below:

```

ensure <acceptance test>
by
  <primary module>
else by
  <alternative module>
else by
  <alternative module>
...
else by
  <alternative module>
else error

```

Like ordinary blocks, recovery blocks can be nested. If a block in a nested recovery block fails its acceptance tests and all its alternatives also fail, then the outer level recovery point will be restored and an alternative module to that block executed.

To show the use of recovery blocks, the various methods used to find the numerical solution of a system of differential equations are considered. As such methods do not give exact solutions, but are subject to various errors, it may be found that some approaches will perform better for certain classes of equations than for others. Unfortunately, methods which give accurate results across a wide range of equations are expensive to implement (in terms of the time needed to complete the method's execution). For example, an **explicit Kutta** method will be more efficient than an **implicit Kutta** method. However, it will only give an acceptable error tolerance for particular problems. There is a class of equations called **stiff** equations whose solution using an explicit Kutta leads to an accumulation of rounding errors; the more expensive implicit Kutta method can more adequately deal with this problem. The following illustrates an approach using recovery blocks which enables the cheaper method to be employed for non-stiff equations but which does not fail when stiff equations are given.

```
ensure rounding_error_within_acceptable_tolerance
by
  Explicit Kutta Method
else by
  Implicit Kutta Method
else error
```

In this example, the cheaper explicit method is usually used; however, when it fails the more expensive implicit method is employed. Although this error is anticipated, this approach also gives tolerance to an error in the design of the explicit algorithm. If the algorithm itself is in error and the acceptance test is general enough to detect both types of error result, the implicit algorithm will be used. When the acceptance test cannot be made general enough, nested recovery blocks can be used. In the following, full design redundancy is provided; at the same time the cheaper algorithm is always used if possible.

```
ensure rounding_error_within_acceptable_tolerance
by
  ensure sensible_value
  by
    Explicit Kutta Method
  else by
    Predictor-Corrector K-step Method
  else error
else by
  ensure sensible_value
  by
    Implicit Kutta Method
  else by
    Variable Order K-Step Method
  else error
else error
```

In the above, two explicit methods are given; when both methods fail to produce a sensible result, the implicit Kutta method is executed. The implicit Kutta method will, of course, also be executed if the value produced by the explicit methods is sensible but not within the required tolerance. Only if all four methods fail will the equations remain unsolved.

The recovery block could have been nested the other way around as shown below. In this case, different behaviour will occur when a non-sensible result is also not within acceptable tolerance. In the first case, after executing the explicit Kutta algorithm, the Predictor Corrector method would be attempted. In the second, the implicit Kutta algorithm would be executed.

```

ensure sensible_value
by
  ensure rounding_error_within_acceptable_margin
  by
    Explicit Kutta Method
  else by
    Implicit Kutta Method
  else error
else by
  ensure rounding_error_within_acceptable_margin
  by
    Predictor-Corrector K-step Method
  else by
    Variable Order K-Step Method
  else error
else error

```

2.6.1 The acceptance test

The acceptance test provides the error detection mechanism which then enables the redundancy in the system to be exploited. The design of the acceptance test is crucial to the efficacy of the recovery block scheme. As with all error detection mechanisms, there is a trade-off between providing comprehensive acceptance tests and keeping the overhead this entails to a minimum, so that normal fault-free execution is affected as little as possible. Note that the term used is **acceptance** not **correctness**; this allows a component to provide a degraded service.

All the error detection techniques discussed in Section 2.5.1 can be used to form the acceptance tests. However, care must be taken in their design as a faulty acceptance test may lead to residual errors going undetected.

2.7 A comparison between *N*-version programming and recovery blocks

Two approaches to providing fault-tolerant software have been described: *N*-version programming and recovery blocks. They clearly share some aspects of their basic philosophy, and yet at the same time they are quite different. This section briefly reviews and compares the two.

- **Static versus dynamic redundancy** – *N*-version programming is based on static redundancy; all versions run in parallel irrespective of whether or not a fault occurs. In contrast, recovery blocks are dynamic in that alternative modules only execute when an error has been detected.
- **Associated overheads** – both *N*-version programming and recovery blocks incur extra development cost, as both require alternative algorithms to be developed.

In addition, for N -version programming, the driver process must be designed and recovery blocks require the design of the acceptance test.

At run-time, N -version programming in general requires N times the resources of a single version. Although recovery blocks only require a single set of resources at any one time, the establishment of recovery points and the process of state restoration is expensive. However, it is possible to provide hardware support for the establishment of recovery points (Lee et al., 1980), and state restoration is only required when a fault occurs.

- **Diversity of design** – both approaches exploit diversity in design to achieve tolerance of unanticipated errors. Both are, therefore, susceptible to errors that originate from the requirements specification.
- **Error detection** – N -version programming uses vote comparison to detect errors whereas recovery blocks use an acceptance test. Where exact or inexact voting is possible there is probably less associated overhead than with acceptance tests. However, where it is difficult to find an inexact voting technique, where multiple solutions exist or where there is a consistent comparison problem, acceptance tests may provide more flexibility.
- **Atomicity** – backward error recovery is criticized because it cannot undo any damage which may have occurred in the environment. N -version programming avoids this problem because all versions are assumed not to interfere with each other: they are atomic. This requires each version to communicate with the driver process rather than directly with the environment. However, it is entirely possible to structure a program such that unrecoverable operations do not appear in recovery blocks.

It perhaps should be stressed that although N -version programming and recovery blocks have been described as competing approaches, they also can be considered as complementary ones. For example, there is nothing to stop a designer using recovery blocks within each version of an N -version system.

2.8 Dynamic redundancy and exceptions

In this section, a framework for implementing software fault tolerance is introduced which is based on dynamic redundancy and the notion of exceptions and exception handlers.

So far in this chapter, the term ‘error’ has been used to indicate the manifestation of a fault, where a fault is a deviation from the specification of a component. These errors can be either anticipated, as in the case of an out of range sensor reading due to hardware malfunction, or unanticipated, as in the case of a design error in the component. An **exception** can be defined as the occurrence of an error. Bringing an exception condition to the attention of the invoker of the operation which caused the exception is called **raising** (or **signalling** or **throwing**) the exception and the invoker’s response is called **handling** (or **catching**) the exception. Exception handling can be considered a *forward error recovery* mechanism, as when an exception has been raised the system is not rolled back to a previous state; instead, control is passed to the handler so that recovery procedures can be initiated. However, as will be shown

in Section 3.4, the exception-handling facility can be used to provide backward error recovery.

Although an exception has been defined as the occurrence of an error, there is some controversy as to the true nature of exceptions and when they should be used. For example, consider a software component or module which maintains a compiler symbol table. One of the operations it provides is to look up a symbol. This has two possible outcomes: *symbol present* and *symbol absent*. Either outcome is an anticipated response and may or may not represent an error condition. If the *lookup* operation is used to determine the interpretation of a symbol in a program body, *symbol absent* corresponds to 'undeclared identifier', which is an error condition. If, however, the *lookup* operation is used during the declaration process, the outcome *symbol absent* is probably the normal case and *symbol present*, that is 'duplicate definition', the exception. What constitutes an error, therefore, depends on the context in which the event occurs. However, in either of the above cases it could be argued that the error is not an error of the symbol table component or of the compiler, in that either outcome is an anticipated result and forms part of the functionality of the symbol table module. Therefore neither outcome should be represented as an exception.

Exception-handling facilities were *not* incorporated into programming languages to cater for programmer design errors; however, it will be shown in Section 3.4 how they can be used to do just that. The original motivation for exceptions came from the requirement to handle abnormal conditions arising in the environment in which a program executes. These exceptions could be termed rare events in the functioning of the environment, and it may or may not be possible to recover from them within the program. A faulty valve or a temperature alarm might cause an exception. These are rare events which, given enough time, might well occur and must be tolerated.

Despite the above, exceptions and their handlers will inevitably be used as a general purpose error-handling mechanism. To conclude, exceptions and exception handling can be used to:

- cope with abnormal conditions arising in the environment;
- enable program design faults to be tolerated;
- provide a general-purpose error-detection and recovery facility.

Exceptions are considered in more detail in Chapter 3.

2.8.1 Ideal fault-tolerant system components

Figure 2.8 shows the ideal component from which to build fault-tolerant systems (Anderson and Lee, 1990). The component accepts service requests and, if necessary, calls upon the services of other components before yielding a response. This may be a normal response or an exception response. Two types of fault can occur in the ideal component: those due to an illegal service request, called **interface exceptions**, and those due to a malfunction in the component itself, or in the components required to service the original request. Where the component cannot tolerate these faults, either by forward or backward error recovery, it raises **failure exceptions** in the calling component. Before raising any exceptions, the component must return itself to a consistent state, if possible, in order that it may service any future request.

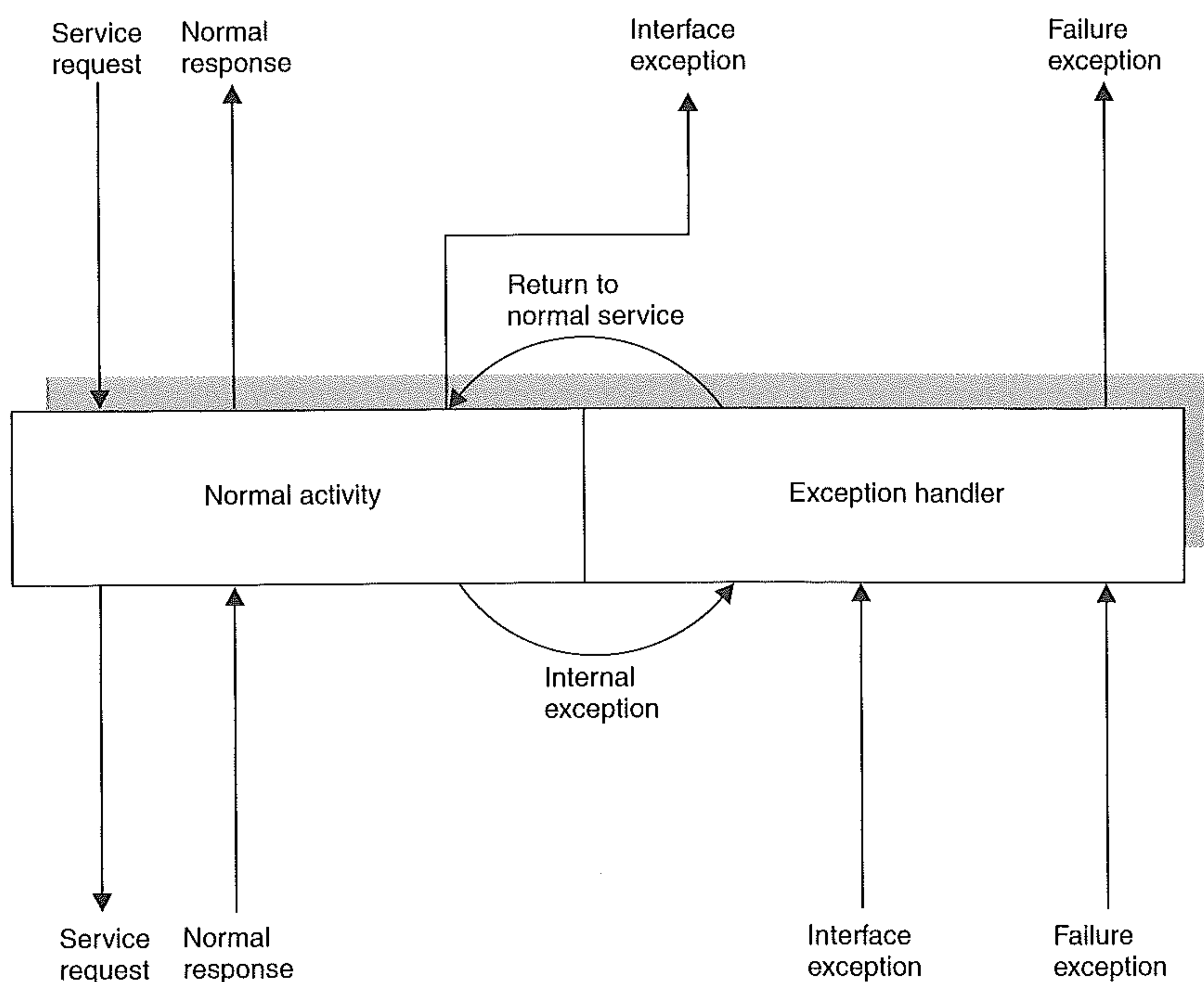


Figure 2.8 An ideal fault-tolerant component.

2.9 Measuring and predicting the reliability of software

Reliability metrics for hardware components have long been established. Traditionally, each component is regarded as a representative of a population of identical members whose reliability is estimated from the proportion of a sample that fail during a specified interval of time, e.g. during testing. Software reliability prediction and measurement, however, is not as well established a discipline. It was ignored for many years by those industries requiring extremely reliable systems because software is assumed not to deteriorate with use; software was regarded as either correct or incorrect – a binary property.

Also, in the past, particular software components were used once only, in the systems for which they were originally intended; consequently, although any errors found during testing were removed, this did not lead to the development of more reliable components which could be used elsewhere. This can be contrasted with hardware components, which are mass produced; any errors found in the design can be corrected, making the next batch more reliable.

The view that software is either correct or not correct is still commonly held. If it is not correct, program testing or program proving will indicate the location of faults which can then be corrected. This chapter has tried to illustrate that the traditional approach of software testing, although indispensable, can never ensure that programs are fault-free, especially with very large and complicated systems where there may be residual specification or design errors. Furthermore, in spite of the continual advances made in the field of proof of correctness, the application of these techniques to non-trivial systems, particularly those involving the concept of time, is still beyond the *state of the art*. Indeed

it may always be beyond the capability of such techniques due to the tendency to make systems and programs ever larger and more complex.

It is for all these reasons that methods of improving reliability through the use of redundancy have been advocated. Unfortunately, even with this approach, it cannot be guaranteed that systems containing software will not fail. It is therefore essential that techniques for assessing software reliability are developed.

As hardware is deemed to be subject to *random* failures it is natural to use a probabilistic approach for reliability assessment. It is perhaps less clear why *systematic* software failures should be characterized similarly. Although systematic in nature, the process by which any particular demand on the system will give rise to a failure is essentially non-deterministic (Littlewood et al., 2001). Software reliability can therefore be considered as *the probability that a given program will operate correctly in a specified environment for a specified length of time*.

Several models have been proposed which attempt to estimate software reliability. These can be broadly classified as (Goel and Bastini, 1985):

- software reliability growth models;
- statistical models.

Growth models attempt to predict the reliability of a program on the basis of its error history (e.g. when faults are identified and repaired). Other statistical models attempt to estimate the reliability of a program by determining its success or failure response to a random sample of test cases, without correcting any errors found. Unfortunately, Littlewood and Strigini (1993) have argued that testing alone can only provide effective evidence for reliability estimates of at best 10^{-4} (that is 10^{-4} failures per hour of operation). This should be compared with the often quoted reliability requirement of 10^{-9} for avionics systems. To increase the assessment of reliability by an order of magnitude to 10^{-5} would require the observation of 460 000 hours (over 50 years) of fault-free operation (Littlewood et al., 2001).

To estimate the reliability of *N*-version components is even more difficult as the level of correlation between the versions is, as indicated earlier, very difficult to estimate. Even strong advocates of the approach would not argue that two 10^{-4} versions would combine to give a 10^{-8} service.

2.10 Safety, reliability and dependability

Safety can be defined as *freedom from those conditions that can cause death, injury, occupational illness, damage to (or loss of) equipment (or property), or environmental harm* (Leveson, 1986). However, as this definition would consider most systems which have an element of risk associated with their use as unsafe, software safety is often considered in terms of **mishaps** (Leveson, 1986). A mishap is an **unplanned event** or **series of events** that can result in death, injury, occupational illness, damage to (or loss of) equipment (or property), or environmental harm.

Although reliability and safety are often considered as synonymous, there is a difference in their emphasis. Reliability has been defined as a measure of the success with which a system conforms to some authoritative specification of its behaviour. This is usually expressed in terms of probability. Safety, however, is the probability that

conditions that can lead to mishaps do not occur *whether or not the intended function is performed*. These two definitions can conflict with each other. For example, measures which increase the likelihood of a weapon firing when required may well increase the possibility of its accidental detonation. In many ways, the only safe aeroplane is one that never takes off; however, it is not very reliable. Nevertheless, any system (or subsystem) whose primary role is to provide safety must itself be *sufficiently* reliable. For example, a secondary Nuclear Reactor Protection System (NRPS) is only required to act when other systems have failed. It provided additional safety and hence is of value if its own reliability is assessed as being no more than 10^{-4} failures per demand. The primary NRPS may be assessed to have reliability of only 10^{-3} ; as long as the primary and secondary systems are independent this provides an overall reliability of at least 10^{-7} . Plant safety is only compromised if these two systems fail *and* the plant controller itself suffers a 'meltdown' failure – an exceedingly rare event in itself.

As with reliability, to ensure the safety requirements of an embedded system, system safety analysis must be performed throughout all stages of its life cycle development. It is beyond the scope of this book to enter into details of safety analysis; for a general discussion of reliability and safety issues, the reader is referred to the Further Reading section at the end of this chapter.

2.10.1 Dependability

The dependability of a system is that property of the system which allows reliance to be justifiably placed on the service it delivers. Dependability, therefore, includes as special cases the notions of reliability, safety and security (Laprie, 1995). Figure 2.9, based on that given by Laprie (1995), illustrates these and other aspects of dependability (where security is viewed in terms of integrity and confidentiality). In this figure, the term 'reliability' is used as a measure of the continuous delivery of a proper service; availability is a measure of the frequency of periods of improper service.

Dependability itself can be described in terms of three components (Laprie, 1995):

- **threats** – circumstances causing or resulting in non-dependability;
- **means** – the methods, tools and solutions required to deliver a dependable service with the required confidence;

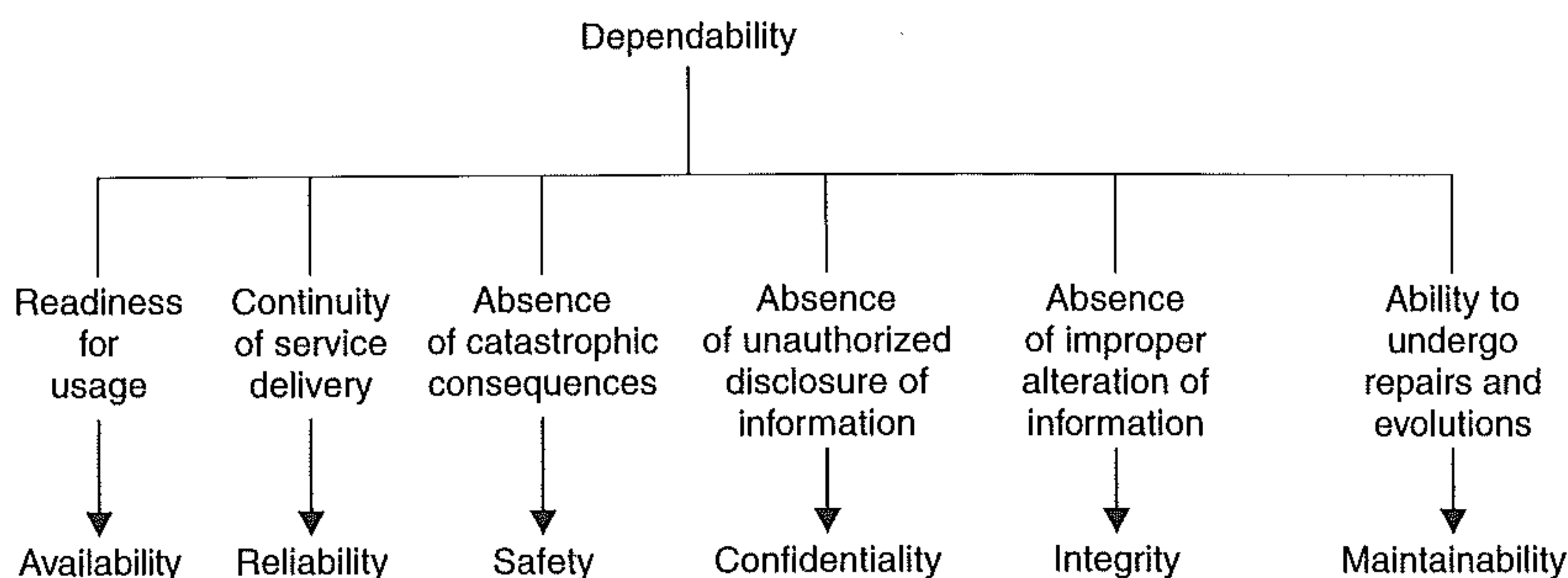


Figure 2.9 Aspects of dependability.

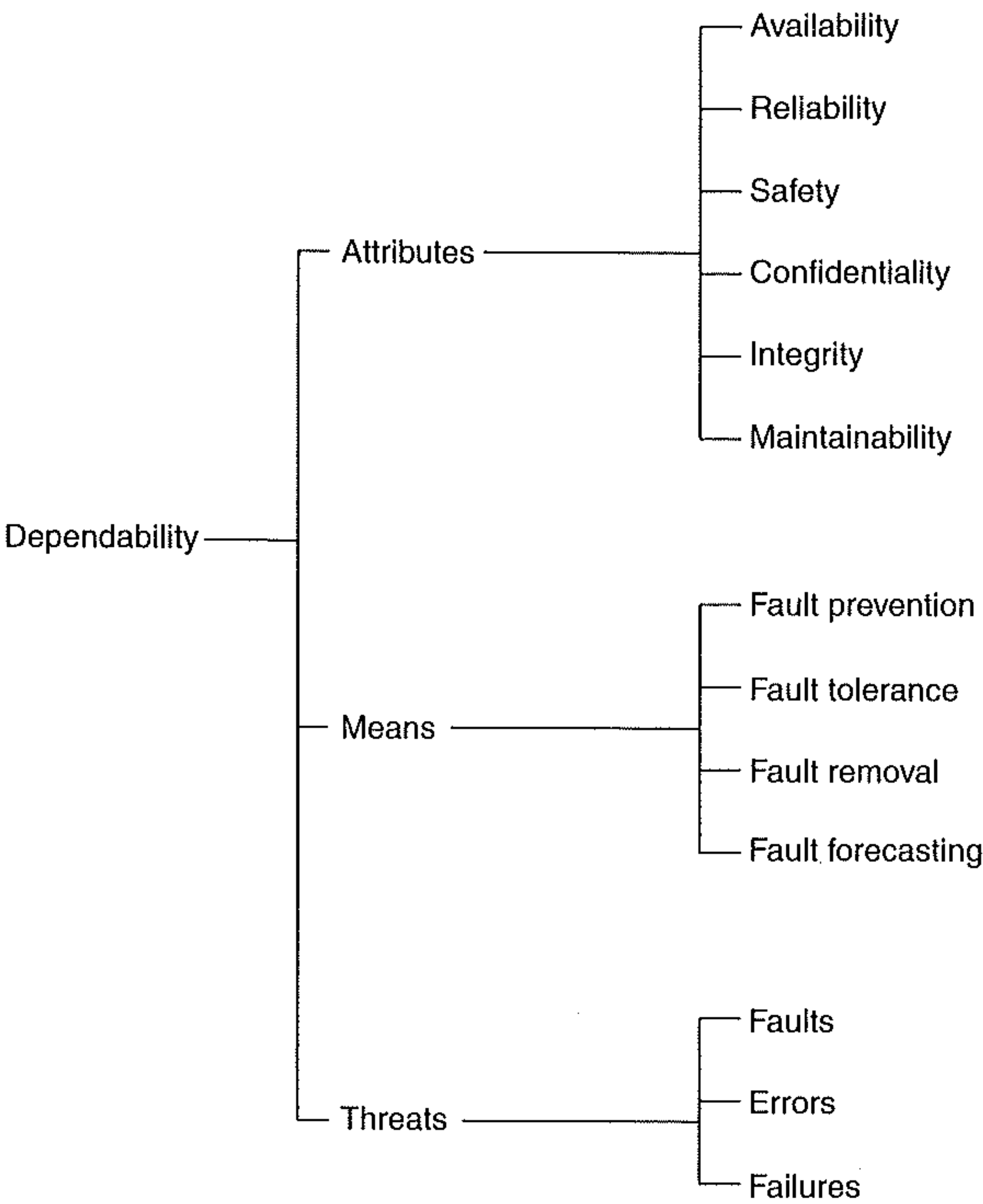


Figure 2.10 Dependability terminology.

- **attributes** – the way and measures by which the quality of a dependable service can be appraised.

Figure 2.10 summarizes the concept of dependability in terms of these three components.

Summary

This chapter has identified reliability as a major requirement for any real-time system. The reliability of a system has been defined as a measure of the success with which the system conforms to some authoritative specification of its behaviour. When the behaviour of a system deviates from that which is specified for it, this is called a failure. Failures result from faults. Faults can be accidentally or intentionally introduced into a system. They can be transient, permanent or intermittent.

There are two approaches to system design which help ensure that potential faults do not cause system failure: fault prevention and fault tolerance. Fault prevention consists of fault avoidance (attempting to limit the introduction of faulty components into the system) and fault removal (the process of finding and removing faults). Fault tolerance involves the introduction of redundant components into a system so that faults can be detected and tolerated. In general, a system will provide either full fault tolerance, graceful degradation or fail-safe behaviour.

Two general approaches to software fault tolerance have been discussed: *N*-version programming (static redundancy) and dynamic redundancy using forward and backward error recovery. *N*-version programming is defined as the independent generation of *N* (where 2 or more) functionally equivalent programs from the same initial specification. Once designed and written, the programs execute concurrently with the same inputs and their results are compared. In principle, the results should be identical, but in practice there may be some difference, in which case the consensus result, assuming there is one, is taken to be correct. *N*-version programming is based on the assumptions that a program can be completely, consistently and unambiguously specified, and that programs which have been developed independently will fail independently. These assumptions may not always be valid, and although *N*-version programming may have a role in producing reliable software it should be used with care and in conjunction with techniques based on dynamic redundancy.

Dynamic redundancy techniques have four constituent phases: error detection, damage confinement and assessment, error recovery, and fault treatment and continued service. One of the most important phases is error recovery for which two approaches have been proposed: backward and forward. With backward error recovery, it is necessary for communicating processes to reach consistent recovery points to avoid the domino effect. For sequential systems, the recovery block has been introduced as an appropriate language concept for expressing backward error recovery. Recovery blocks are blocks in the normal programming language sense except that at the entrance to the block is an automatic recovery point and at the exit an acceptance test. The acceptance test is used to test that the system is in an acceptable state after the execution of the primary module. The failure of the acceptance test results in the program being restored to the recovery point at the beginning of the block and an alternative module being executed. If the alternative module also fails the acceptance test, the program is restored to the recovery point again and yet another module is executed, and so on. If all modules fail then the block fails. A comparison between *N*-version programming and recovery blocks illustrated the similarities and differences between the approaches.

Although forward error recovery is system-specific, exception handling has been identified as an appropriate framework for its implementation. The concept of an ideal fault-tolerant component was introduced which used exceptions.

Finally in this chapter, the notions of software safety and dependability were introduced.

Further reading

- Anderson, T. and Lee, P. A. (1990) *Fault Tolerance, Principles and Practice*, 2nd edn. Englewood Cliffs, NJ: Prentice Hall.
- Andrews, J. D. and Moss, T. R. (2002) *Reliability and Risk Assessment*, 2nd edn. Chichester: Wiley.
- De Florio, V. and Blondia, C. (2008) A survey of linguistic structures for application-level fault tolerance, *ACM Computer Surveys*, **40**(2).

- Herrmann, D. S. (1999) *Software Safety and Reliability*. Los Alamitos, CA: IEEE Computer Society.
- Kritzinger, D. (2006) *Aircraft System Safety – Military and Civil Aeronautical Applications*. Cambridge: Woodhead Publishing.
- Laprie J.-C. et al. (1995) *Dependability Handbook*. Toulouse: Cépaduès (in French).
- Leveson, N. G. (1995) *Safeware: System Safety and Computers*. Reading, MA: Addison-Wesley.
- Mili, A. (1990) *An Introduction to Program Fault Tolerance*. New York: Prentice Hall.
- Neumann, P. G. (1995) *Computer-Related Risks*. Reading, MA: Addison-Wesley.
- Redmill, F. and Rajan, J. (eds) (1997) *Human Factors in Safety-Critical Systems*. Oxford: Butterworth-Heinemann.
- Storey, N. (1996) *Safety-Critical Computer Systems*. Reading, MA: Addison-Wesley.

Exercises

- 2.1 Is a program reliable if it conforms to an erroneous specification of its behaviour?
- 2.2 What would be the appropriate levels of degraded service for a computer-controlled automobile?
- 2.3 Write a recovery block for sorting an array of integers.
- 2.4 To what extent is it possible to detect recovery lines at run-time? (See Anderson and Lee, 1990, Chapter 7.)
- 2.5 Figure 2.11 illustrates the concurrent execution of four communicating processes (P_1 , P_2 , P_3 and P_4) and their associated recovery points (for example, R_{11} is the first recovery point for process P_1).

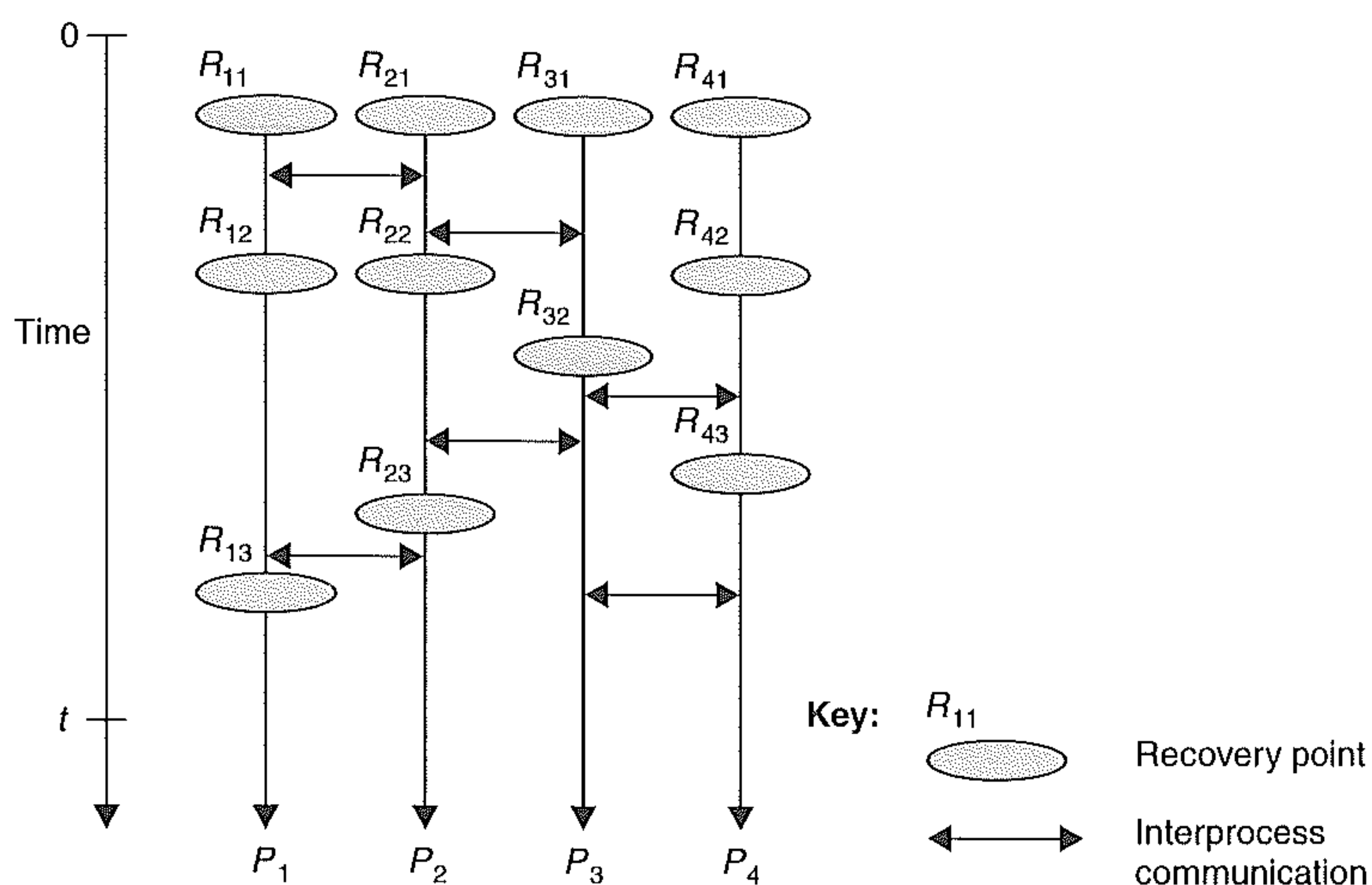


Figure 2.11 Concurrent execution of four processes for Exercise 2.5.

Explain what happens when an error is detected by:

- Process P_1 at time t ;
- Process P_2 at time t .

- 2.6** Should the end of file condition that occurs when sequentially reading a file be signalled to the programmer as an exception?
- 2.7** Data diversity is a fault-tolerance strategy that complements design diversity. Under what conditions might data diversity be more appropriate than design diversity? (Hint: see Ammann and Knight, 1988.)
- 2.8** Should the dependability of a system be judged by an independent assessor?

Chapter 3

Exceptions and exception handling

3.1	Exception handling in older real-time languages	3.4	Recovery blocks and exceptions
3.2	Modern exception handling		Summary
3.3	Exception handling in Ada, Java and C		Further reading
			Exercises

Chapter 2 considered how systems can be made more reliable and introduced exceptions as a framework for implementing software fault tolerance. In this chapter, exceptions and exception handling are considered in more detail and their provision in particular real-time programming languages is discussed.

There are a number of general requirements for an exception-handling facility.

- (R1) As with all language features, the facility must be simple to understand and use.
- (R2) The code for exception-handling should not be so obtrusive as to obscure understanding of the program's normal error-free operation. A mechanism which intermingles code for normal processing and exceptional processing will prove difficult to understand and maintain. It may well lead to a less reliable system.
- (R3) The mechanism should be designed so that run-time overheads are incurred only when handling an exception. Although the majority of applications require that the performance of a program which uses exceptions is not adversely affected under normal operating conditions, this may not always be the case. Under some circumstances, in particular where speed of recovery is of prime importance, an application may be prepared to tolerate a little overhead on the normal error-free operation.
- (R4) The mechanism should allow the uniform treatment of exceptions detected both by the environment and by the program. For example, an exception such as **arithmetic overflow**, which is detected by the hardware, should be handled in exactly the same manner as an exception raised by the program as a result of an assertion failure.
- (R5) As already mentioned in Chapter 2, the exception mechanism should allow recovery actions to be programmed.

3.1 Exception handling in older real-time languages

Although the terms ‘exception’ and ‘exception handling’ are relatively new, they simply express an approach to programming which attempts to contain and handle error situations. Consequently, most programming languages have facilities which enable at least some exceptions to be handled. This section briefly appraises these facilities in terms of the requirements set out above.

3.1.1 Unusual return value and status flags

One of the most primitive forms of an exception-handling mechanism is the *unusual return value* or *error return* from a procedure or a function. Its main advantage is that it is simple and does not require any new language mechanism for its implementation. C supports this approach, and typically it would be used as follows:

```
if(function_call(parameters) == AN_ERROR) {
    /* error handling code */
} else {
    /* normal return code */
};
```

Where a function already returns a value and it is not possible to partition the range of values to indicate an error, then *status flags* are used. These are atomic shared variables which can be set and tested. For example, C/Real-Time POSIX has a shared integer variable called `errno`¹ that is set by the system to the most recent detected error condition. Using this approach, the above code can be rewritten:

```
#include <errno.h>
...
ret = function_call(parameters);
if (errno == AN_ERROR) {
    /* error handling code */
} else {
    /* normal return code */
};
```

As can be seen, although this meets the simplicity requirement R1 and allows recovery actions to be programmed (R5), it fails to satisfy R2, R3 and R4. The code is obtrusive, it entails overheads every time it is used, and it is not clear how to handle errors detected by the environment. Furthermore, if the error is not checked for, the program can fail in an unpredictable manner.

Throughout this book, C is used in conjunction with Real-Time POSIX. Error conditions from Real-Time POSIX are indicated by a combination of return of non-zero values (each with a symbolic name) and use of the `errno` facility. For reliability, every call to a system function should test to ensure no unexpected errors have occurred. However, as illustrated above, this can obscure the structure of the code. Consequently for pedagogical purposes, this book will use a stylized interface to C/Real-Time POSIX. For every POSIX system call, it is assumed that there is a macro defined which undertakes

¹In a multithread environment, `errno` is a function that returns a thread-specific error code.

an error check. For example, a system call named `sys_call` which takes one parameter will have the following macro automatically defined.

```
#define SYS_CALL(A) if(sys_call(A) != 0) error()

/* or */

#define SYS_CALL(A) sys_call(A); if (errno != 0) error()
```

where `error` is a function which undertakes error processing. Hence the code shown will simply be `SYS_CALL(param)`.

3.1.2 Forced branch

In assembly languages, the typical mechanism for exception handling is for subroutines to *skip return*. In other words, the instruction immediately following the subroutine call is skipped to indicate the presence (or the absence) of an error. This is achieved by the subroutine incrementing its return address (program counter) by the length of a simple jump instruction to indicate an error-free (or error) return. In the case where more than one exceptional return is possible, the subroutine will assume that the caller has more than one jump instruction after the call, and will manipulate the program counter accordingly.

For example, assuming two possible error conditions, the following might be used to call a subroutine which outputs a character to a device.

```
jsr pc, PRINT_CHAR
jmp IO_ERROR
jmp DEVICE_NOT_ENABLED
# normal processing
```

The subroutine, for a normal return, would increment the return address by two `jmp` instructions.

Although this approach incurs little overhead (R3) and enables recovery actions to be programmed (R5), it can lead to obscure program structures, and therefore violates requirements R1 and R2. R4 also cannot be satisfied.

3.1.3 Non-local `goto` and error procedures

A high-level language version of a forced branch might require different labels to be passed as parameters to procedures or to have standard label variables (a label variable is an object to which a program address can be assigned and which can be used to transfer control). RTL/2 is an example of an early real-time language which provides the latter facility in the form of a non-local `goto`.

Notice that, when used in this way, the `goto` is more than just a jump; it implies an abnormal return from a procedure. Consequently, the stack must be unwound until the environment restored is that of the procedure containing the declaration of the label. The equivalent result can be obtained in the C language by using the `setjmp` and `longjmp` facility. The `setjmp` establishes the label and the `longjmp` performs the `goto`.

With this approach, the penalty of unwinding the stack is only incurred when an error has occurred, so requirement R3 has been satisfied. Although the use of `gotos`

is very flexible (satisfying R4 and R5), they can lead to very obscure programs. They therefore fail to satisfy the requirements R1 and R2.

Using a non-local `goto`, the control flow of the program has been broken. This is appropriate for unrecoverable errors. For recoverable errors an *error procedure variable* can be used. Again, the main criticism of this approach is that programs can become very difficult to understand and maintain.

3.2 Modern exception handling

It has been shown that the traditional approaches to exception handling often result in the handling code being intermingled with the program's normal flow of execution. The modern approach is to introduce exception-handling facilities directly into the language and thereby provide a more structured exception-handling mechanism. The exact nature of these facilities varies from language to language; however, there are several common threads that can be identified. These are discussed in the following subsections.

3.2.1 Exceptions and their representation

In Section 2.5.1, it was noted that there are two types of error detection technique: environmental detection and application detection. Also, depending on the delay in detecting the error, it may be necessary to raise the exception synchronously or asynchronously. A synchronous exception is raised as an immediate result of a section of code attempting an inappropriate operation. An asynchronous exception is raised some time after the operation that resulted in the error occurring. It may be raised in the process that originally executed the operation or in another process. There are therefore four classes of exceptions.

- (1) Detected by the environment and raised synchronously – an array bounds violation or divide by zero are examples of such exceptions.
- (2) Detected by the application and raised synchronously – for example, the failure of a program-defined assertion check.
- (3) Detected by the environment and raised asynchronously – an exception raised as the result of power failure or the failure of some health-monitoring mechanism.
- (4) Detected by the application and raised asynchronously – for example, one process may recognize that an error condition has occurred that will result in another process not meeting its deadline or not terminating correctly.

Asynchronous exceptions are often called asynchronous notifications or signals and are usually considered in the context of concurrent programming. This chapter will therefore focus on synchronous exception handling and leave the topic of asynchronous exception handling until Chapter 7.

With synchronous exceptions, there are several models for their declaration. For example, they can be viewed as:

- a constant name which needs to be explicitly declared; or
- an object of a particular type which may or may not need to be explicitly declared.

Program 3.1 Predeclared exceptions in package `Standard`.

```

package Standard is
  ...
  Constraint_Error : exception;
  Program_Error   : exception;
  Storage_Error   : exception;
  Tasking_Error   : exception;
  ...
end Standard;

```

Ada requires exceptions to be declared like constants; for example, the exceptions that can be raised by the Ada run-time environment are declared in package `Standard` – see Program 3.1. This package is visible to all Ada programs.

Java and C++ take a more object-oriented view of exceptions. In Java, exceptions are objects whose class is a subclass of the `Throwable` class. They can be **thrown** by the run-time systems and the application (see Section 3.3.2). In C++, exceptions of any object type may be thrown without predeclaration.

3.2.2 The domain of an exception handler

Within a program, there may be several handlers for a particular exception. Associated with each handler is a **domain** which specifies the region of computation during which, if an exception occurs, the handler will be activated. The accuracy with which a domain can be specified will determine how precisely the source of the exception can be located. In a block structured language, like Ada, the domain is normally the block. For example, consider a temperature sensor whose value should fall in the range 0 to 100°C. The following Ada block defines temperature to be an integer between 0 and 100. If the calculated value falls outside that range, the run-time support system for Ada raises a `Constraint_Error` exception. The invocation of the associated handler enables any necessary corrective action to be performed.

```

declare
  subtype Temperature is Integer range 0 .. 100;
begin
  -- read temperature sensor and calculate its value
exception
  -- handler for Constraint_Error
end;

```

The Ada details will be filled in shortly.

Where blocks form the basis of other units, such as procedures and functions, the domain of an exception handler is usually that unit.

In some languages, such as Java, C++ and Modula-3, not all blocks can have exception handlers. Rather, the domain of an exception handler must be explicitly indicated and the block is considered to be **guarded**; in Java this is done using a ‘try-block’:

```

try {
    // statements which may raise exceptions
}
catch (ExceptionType e) {
    // handler for e
}

```

As the domain of the exception handler specifies how precisely the error can be located, it can be argued that the granularity of the block is inadequate. For example consider the following sequence of calculations in Ada, each of which possibly could cause `Constraint_Error` to be raised:

```

declare
    subtype Temperature is Integer range 0 .. 100;
    subtype Pressure is Integer range 0 .. 50;
    subtype Flow is Integer range 0 .. 200;
begin
    -- read temperature sensor and calculate its value
    -- read pressure sensor and calculate its value
    -- read flow sensor and calculate its value

    -- adjust temperature, pressure and flow
    -- according to requirements
exception
    -- handler for Constraint_Error
end;

```

The problem for the handler is to decide which calculation caused the exception to be raised. Further difficulties arise when arithmetic overflow and underflow can occur.

With exception handler domains based on blocks, one solution to this problem is to decrease the size of the block and/or nest them. Using the sensor example:

```

declare
    subtype Temperature is Integer range 0 .. 100;
    subtype Pressure is Integer range 0 .. 50;
    subtype Flow is Integer range 0 .. 200;
begin
    begin
        -- read temperature sensor and calculate its value
    exception
        -- handler for Constraint_Error for temperature
    end;
    begin
        -- read pressure sensor and calculate its value
    exception
        -- handler for Constraint_Error for pressure
    end;
    begin
        -- read flow sensor and calculate its value
    exception
        -- handler for Constraint_Error for flow
    end;
    -- adjust temperature, pressure and flow according
    -- to requirements

```



```

exception
  -- handler for other possible exceptions
end;
```

Alternatively, procedures containing handlers could be created for each of the nested blocks. However, in either case this can become long-winded and tedious. A different solution is to allow exceptions to be handled at the statement level. Using such an approach the above example would be rewritten thus:

```

-- NOT VALID Ada declare
  subtype Temperature is Integer range 0 .. 100;
  subtype Pressure is Integer range 0 .. 50;
  subtype Flow is Integer range 0 .. 200;
begin
  Read_Temperature_Sensor;
    exception -- handler for Constraint_Error;

  Read_Pressure_Sensor;
    exception -- handler for Constraint_Error;

  Read_Flow_Sensor;
    exception -- handler for Constraint_Error;

  -- adjust temperature, pressure and flow
  -- according to requirements

end;
```

The CHILL programming language (CCITT, 1980) has such a facility. Although this enables the cause of the exception to be located more precisely, it intermingles the exception-handling code with the normal flow of operation, which may result in less clear programs and violate requirement R2 (given at the beginning of this chapter).

The preferred approach to this problem is to allow parameters to be passed with the exceptions. With Java this is automatic, as the exception is an object, and therefore can contain as much information as the programmer wishes. In contrast, Ada provides a pre-defined procedure `Exception_Information` that returns implementation-defined details on the occurrence of the exception.

3.2.3 Exception propagation

Closely related to the concept of an exception domain is the notion of exception propagation. So far it has been implied that if a block or procedure raises an exception, then there is a handler associated with that block or procedure. However, this may not be the case, and there are two possible methods for dealing with a situation where no immediate exception handler can be found.

The first approach is to regard the absence of a handler as a programmer error which should be reported at compile-time. However, it is often the case that an exception raised in a procedure can only be handled within the context from which the procedure was called. In this situation, it is not possible to have the handler local to the procedure. For example, an exception raised in a procedure as a result of a failed assertion involving the parameters can only be handled in the calling context. Unfortunately, it is not always

possible for the compiler to check whether the calling context includes the appropriate exception handlers, as this may require complex flow control analysis. This is particularly difficult when the procedure calls other procedures which may also raise exceptions. Consequently, languages which need compile-time error generation for such situations require that a procedure specifies which exceptions it may raise (that is, not handled locally). The compiler can then check the calling context for an appropriate handler and if necessary generate the required error message. This is the approach taken by the CHILL language. Java and C++ also allow a function to define which exceptions it can raise. However, unlike CHILL, they do not require a handler to be available in the calling context.

The second approach, which can be adopted when no local handler for an exception can be found, is to look for handlers up the chain of invokers at run-time; this is called **propagating** the exception. Ada and Java allow exception propagation, as do languages such as C++ and Modula-2/3.

A potential problem with exception propagation occurs when the language requires exceptions to be declared and thus given scope. Under some circumstances it is possible for an exception to be propagated outside its scope, thereby making it impossible for a handler to be found. To cope with this situation, most languages provide a 'catch all' exception handler. This handler is also used to save the programmer enumerating many exception names.

An unhandled exception causes a sequential program to be aborted. If the program contains more than one process and a particular process does not handle an exception it has raised, then usually that process is aborted. However, it is not clear whether the exception should be propagated to the parent process. Exceptions in multi-process programs will be considered in detail in Chapter 7.

Another way of considering the exception propagation issue is in terms of whether the handlers are statically or dynamically associated with exceptions. Static association, as in CHILL, is done at compile-time, and therefore cannot allow propagation because the chain of invokers is unknown. Dynamic association is performed at run-time, and therefore can allow propagation. Although dynamic association is more flexible, it does entail more run-time overhead as the handler must be searched for; with static association a compile-time address can be generated.

3.2.4 Resumption versus termination model

A crucial consideration in any exception-handling facility is whether the invoker of the exception should continue its execution after the exception has been handled. If the invoker can continue, then it may be possible for the handler to cure the problem that caused the exception to be raised and for the invoker to resume as if nothing has happened. This is referred to as the **resumption** or **notify** model. The model where control is not returned to the invoker is called **termination** or **escape**. Clearly it is possible to have a model in which the handler can decide whether to resume the operation which caused the exception, or to terminate the operation. This is called the **hybrid** model.

The resumption model

To illustrate the resumption model, consider three procedures P , Q and R . Procedure P invokes Q which in turn invokes R . Procedure R raises an exception r which is handled by Q , assuming there is no local handler in R . The handler for r is Hr . In the course of

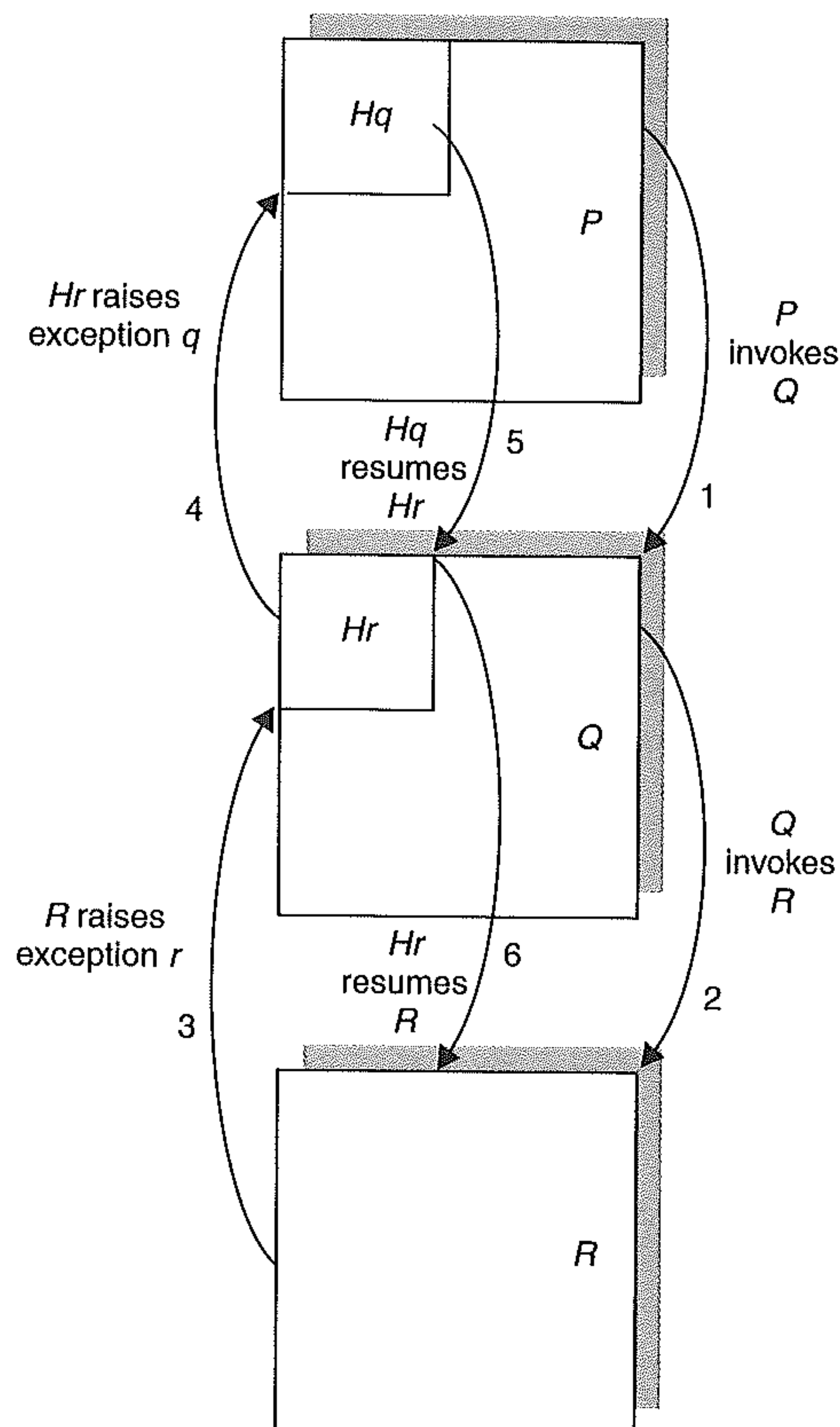


Figure 3.1 The resumption model.

handling r , Hr raises exception q which is handled by Hq in procedure P (the caller of Q). Once this has been handled Hr continues its execution and when finished R continues. Figure 3.1 represents this sequence of events diagrammatically by arcs numbered 1 to 6.

The resumption model is most easily understood by viewing the handler as an implicit procedure which is called when the exception is raised.

The problem with this approach is that it is often difficult to repair errors which are raised by the run-time environment. For example, an arithmetic overflow occurring in the middle of a sequence of complex expressions may result in several registers containing partial evaluations. As a consequence of calling the handler, these registers may be overwritten.

The languages Pearl and Mesa both provide a mechanism whereby a handler can return to the context from which the exception was raised. Both languages also support the termination model.

Although implementing a strict resumption model is difficult, a compromise is to re-execute the block associated with the exception handler. The Eiffel language (Meyer, 1992) provides such a facility, called **retry**, as part of its exception-handling model. The handler is able to set a local flag to indicate that an error has occurred and the block is able to test that flag. Note that for such a scheme to work, the local variables of the block must not be re-initialized on a retry.

The advantage of the resumption model comes when the exception has been raised asynchronously, and therefore has little to do with the current process's execution. Asynchronous event handling is discussed in detail in Section 7.4.

The termination model

In the termination model, when an exception has been raised and the handler has been called, control does not return to the point where the exception occurred. Instead, the block or procedure containing the handler is terminated, and control passed to the calling block or procedure. An invoked procedure may therefore terminate in one of a number of conditions. One of these is the **normal condition**, while the others are **exception conditions**.

When the handler is inside a block, control is given to the first statement following the block after the exception has been handled; as the following example shows.

```
declare
  subtype Temperature is Integer range 0 .. 100;
begin
  ...
  begin
    -- read temperature sensor and calculate its value,
    -- may result in an exception being raised
  exception
    -- handler for Constraint_Error for temperature,
    -- once handled this block terminates
  end;

  -- code here executed when block exits normally
  -- or when an exception has been raised and handled.

exception
  -- handler for other possible exceptions
end;
```

With procedures, as opposed to blocks, the flow of control can change quite dramatically, as Figure 3.2 illustrates. Again procedure *P* has invoked procedure *Q*, which has in turn called procedure *R*. An exception is raised in *R* and handled in *Q*.

Ada, Java, C++, Modula-2/3 and CHILL all have the termination model of exception handling.

The hybrid model

With the hybrid model, it is up to the handler to decide whether the error is recoverable. If it is, the handler can return a value and the semantics are the same as in the resumption model. If the error is not recoverable, the invoker is terminated. The signal mechanisms of Mesa and Real-Time Basic (Bull and Lewis, 1983) provide such a facility. As noted before, Eiffel also supports the restricted 'retry' model.

Exception handling and operating systems

In many cases, a program in a language like Ada or Java will be executed on top of an operating system such as Linux or XP. These systems will detect certain synchronous error conditions; for example, memory violation or illegal instruction. Typically, this will result

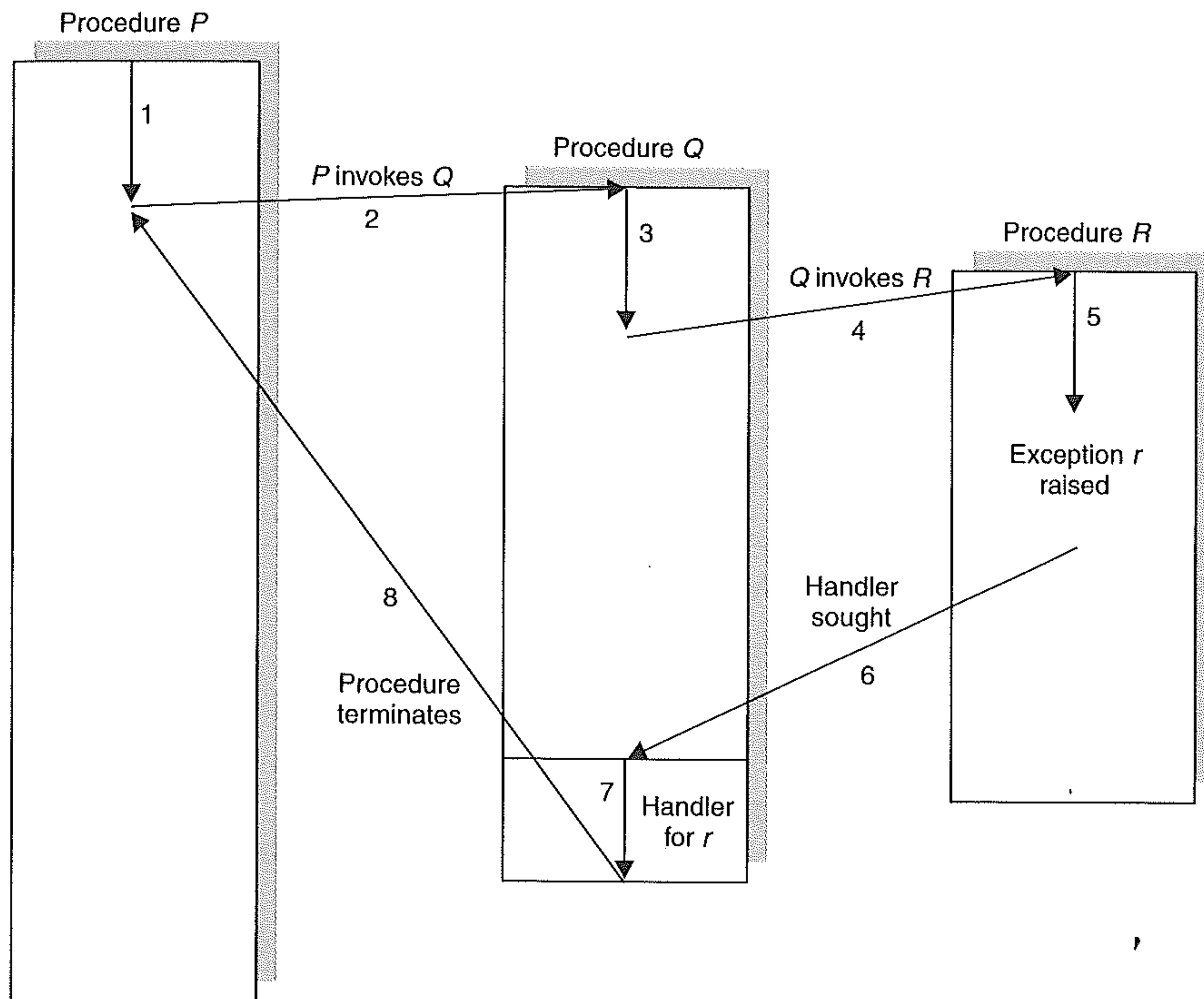


Figure 3.2 The termination model.

in the executing process being terminated. However, many systems allow the programmer to attempt error recovery. The recovery model supported by C/Real-Time POSIX, for instance, allows the programmer to handle these synchronous exceptions (via *signals*) by associating a handler with the exception. This handler is called by the system when the error condition is detected. Once the handler is finished, the process is resumed at the point where it was 'interrupted' – hence C/Real-Time POSIX supports the resumption model.

If a language supports the termination model, it is the responsibility of the run-time support system for that language to catch the error and undertake the necessary manipulation of program state so that the programmer can use the termination model.

C/Real-Time POSIX signals will be considered in detail in Chapter 7, as they are really an asynchronous concurrency mechanism.

3.3 Exception handling in Ada, Java and C

Exception handling in sequential Ada, Java and C is now considered. The three languages all have different philosophies. Exception handling in concurrent systems will be described in Chapter 7.

3.3.1 Ada

The Ada language supports explicit exception declaration, the termination model of exception handling with propagation of unhandled exceptions, and a limited form of exception parameters.

Exception declaration

Exceptions in Ada are declared in the same fashion as constants; the type of constant being defined by the keyword **exception**. The following example declares an exception called `Stuck_Valve`.

```
Stuck_Valve : exception;
```

Every exception declared using the keyword **exception** has an associated `Exception_Id` that is supported by the predefined package `Ada.Exceptions` (see Program 3.2). This identity can be obtained using the predefined attribute `Identity`. The identity of the `Stuck_Valve` exception, given above, can be found by:

```
with Ada.Exceptions;
with Valves; -- for example
package My_Exceptions is
  Id : Ada.Exceptions.Exception_Id :=
    Valves.Stuck_Valve'Identity;
end My_Exceptions;
```

assuming that `Stuck_Valve` is declared in package `Valves`.

An exception can be declared in the same place as any other declaration and, like every other declaration, it has scope.

The language has several standard exceptions whose scopes are the whole program. These exceptions may be raised by the language's run-time support system in response to certain error conditions. They include:

- `Constraint_Error` – this is raised, for example, when an attempt is made to assign a value to an object which is outside its declared range, when an access to an array is outside the array bounds, or when access using a null pointer is attempted. It is also raised by the execution of a predefined numeric operation that cannot deliver a correct result within the declared accuracy for real types. This includes the familiar divide by zero error.
- `Storage_Error` – this is raised when the dynamic storage allocator is unable to fulfil a demand for storage because the physical limitations of the machine have been exhausted.

Raising an exception

As well as exceptions being raised by the environment in which the program executes, they may also be raised explicitly by the program using the **raise** statement. The following example raises the exception `Io_Error` (which must have been previously declared and be in scope) if an I/O request produces device errors.

```
begin
  ...
  -- statements that request a device to perform some I/O
  if Io_Device_In_Error then
    raise Io_Error;
  end if;
  ...
end;
```

Program 3.2 The Ada.Exceptions package.

```

package Ada.Exceptions is
  type Exception_Id is private;
    -- each exception has an associated identifier
  Null_Id : constant Exception_Id;
  function Exception_Name(Id : Exception_Id) return String;
    -- returns the name of the object which
    -- has the Exception_Id Id

  type Exception_Occurrence is limited private;
    -- each exception occurrence has an associated identifier
  type Exception_Occurrence_Access is
    access all Exception_Occurrence;
  Null_Occurrence : constant Exception_Occurrence;

  procedure Raise_Exception(E : in Exception_Id;
    Message : in String := "");
    -- raises the exception E and associates Message with
    -- the exception occurrence

  function Exception_Message(X : Exception_Occurrence)
    return String;
    -- allows the string passed by Raise_Exception to be accessed,
    -- in the handler; if the exception was raised by the raise
    -- statement, the string contains implementation-defined
    -- information about the exception

  procedure Reraise_Occurrence(X : in Exception_Occurrence);
    -- reraises the exception identified by the exception
    -- occurrence parameter

  function Exception_Identity(X : Exception_Occurrence)
    return Exception_Id;
    -- returns the exception identifier of the exception
    -- occurrence passed as a parameter

  function Exception_Name(X : Exception_Occurrence)
    return String;
    -- same as Exception_Name(Exception_Identity(X)).

  function Exception_Information(X : Exception_Occurrence)
    return String;
    -- the same as Exception_Message(X) but contains more details
    -- if the message comes from the implementation

  procedure Save_Occurrence(Target : out Exception_Occurrence;
    Source : in Exception_Occurrence);
    -- allows assignment to objects of type Exception_Occurrence

  function Save_Occurrence(Source : Exception_Occurrence)
    return Exception_Occurrence_Access;
    -- allows assignment to objects of type Exception_Occurrence
private
  ... -- not specified by the language
end Ada.Exceptions;

```

Notice that no `else` part of the `if` statement is required because control is *not* returned to the statement following the `raise`.

If `Io_Error` had been declared as an `Exception_Id`, it would have been necessary to raise the exception using the procedure `Ada.Exceptions.Raise_Exception`. This would have allowed a textual string to be passed as a parameter to the exception.

Each individual raising of an exception is called an exception **occurrence** and is represented by a value of type `Ada.Exceptions.Exception_Occurrence`. When an exception is handled, the value of the `Exception_Occurrence` can be found and used to determine more information about the cause of the exception.

Exception handling

Every block in Ada (and every subprogram, accept statement or task) can contain an optional collection of exception handlers. These are declared at the end of the block (or subprogram, accept statement or task). Each handler is a sequence of statements. Preceding the sequence are: the keyword **when**, an optional parameter (to which the identity of the exception occurrence will be assigned), the names of the exceptions which are to be serviced by the handler, and the symbol `=>`. For example, the following block declares three exceptions and provides two handlers.

```
declare
  Sensor_High, Sensor_Low, Sensor_Dead : exception;
  -- other declarations
begin
  -- statements that may cause the above exceptions
  -- to be raised
exception
  when E: Sensor_High | Sensor_Low =>
    -- Take some corrective action
    -- if either Sensor_High or Sensor_Low is raised.
    -- E contains the exception occurrence
  when Sensor_Dead =>
    -- sound an alarm if the exception
    -- Sensor_Dead is raised
end;
```

To avoid enumerating all possible exception names, Ada provides a **when others** handler name. This is only allowed as the last exception-handling choice and stands for all exceptions not previously listed in the current collection of handlers. For example, the following block prints out information about the exception, and sounds an alarm when any exception except `Sensor_Low` or `Sensor_High` is raised (including `Sensor_Dead`).

```
declare
  Sensor_High, Sensor_Low, Sensor_Dead : exception;
  -- other declarations
  use Text_Io;
begin
  -- statements that may cause the above exceptions
  -- to be raised
```



```

exception
  when Sensor_High | Sensor_Low =>
    -- take some corrective action
  when E: others =>
    Put(Exception_Name(E));
    Put_Line(" caught. The following information is available ");
    Put_Line(Exception_Information(E));
    -- sound an alarm
end;

```

An exception raised in an exception handler cannot be handled by that handler or other handlers in the same block (or procedure). Instead, the block is terminated and a handler sought in the surrounding block or at the point of call for a subprogram.

Exception propagation

If there is no exception handler in the enclosing block (or subprogram or accept statement), the exception is raised again. Ada thus **propagates** exceptions. In the case of a block, this results in the exception being raised in the enclosing block, or subprogram. In the case of a subprogram, the exception is raised at its point of call.

A common misconception with Ada is that exception handlers can be provided in the initialization section of packages to handle exceptions that are raised in the execution of their nested subprograms. An exception raised and *not* handled by a subprogram is propagated to the caller of the subprogram. Therefore, such an exception will only be handled by the initialization code if it itself called the subprogram. The following example illustrates this point.

```

package Temperature_Control is
  subtype Temperature is Integer range 0 .. 100;
  Sensor_Dead, Actuator_Dead : exception;

  procedure Set_Temperature(New_Temp : in Temperature);
    -- raises Actuator_Dead
  function Read_Temperature return Temperature;
    -- raises Sensor_Dead
end Temperature_Control;

package body Temperature_Control is
  procedure Set_Temperature(New_Temp : in Temperature) is
  begin
    -- inform actuator of new temperature
    if No_Response then
      raise Actuator_Dead;
    end if;
  end Set_Temperature;

  function Read_Temperature return Temperature is
  begin
    -- read sensor
    if No_Response then
      raise Sensor_Dead;
    end if;
    -- calculate temperature
    return Reading;
  end Read_Temperature;
end Temperature_Control;

```

```

exception
  when Constraint_Error =>
    -- the temperature has gone outside
    -- its expected range;
    -- take some appropriate action
end Read_Temperature;
begin
  -- initialization of package
  Set_Temperature(Initial_Reading);
exception
  when Actuator_Dead =>
    -- take some corrective action
end Temperature_Control;

```

In this example, the procedure `Set_Temperature`, which can be called from outside the package, is also called during the initialization of the package. This procedure may raise the exception `Actuator_Dead`. The handler for `Actuator_Dead` given in the initialization section of the package *will only catch the exception when the procedure is called from the initialization code*. It will not catch the exception when the procedure is called from outside the package.

If the code which initialized a package body itself raises an exception which is not handled locally, the exception is propagated to the point where the package came into scope.

Last wishes

An exception can also be propagated by a program re-raising the exception in the local handler. The statement **raise** (or the procedure `Ada.Exceptions.Reraise_Occurrence`) has the effect of re-raising the last exception (or the specific exception occurrence). This facility is useful in the programming of **last wishes**. Here it is often the case that the significance of an exception is unknown to the local handler, but must be handled in order to clean up any partial resource allocation that may have occurred previous to the exception being raised. For example, consider a procedure that allocates several devices. Any exception raised during the allocation routine, which is propagated directly to the caller, may leave some devices allocated. The allocator, therefore, wishes to deallocate the associated devices if it has not been possible to allocate the full request. The following illustrates this approach.

```

subtype Devices is Integer range 1 .. Max;

procedure Allocate (Number : Devices) is
begin
  -- request each device be allocated in turn
  -- noting which requests are granted
exception
  when others =>
    -- deallocate those devices allocated
    raise; -- re-raise the exception
end Allocate;

```

Used in this way, the procedure can be considered to implement the failure atomicity property of an atomic action; all the resources are allocated or none are (see Chapter 7).

As a further illustration, consider a procedure which sets the positions of slats and flaps on the wings of a fly-by-wire aircraft during its landing phase. These alter the amount of lift on the plane; asymmetrical wing settings on landing (or take-off) will cause the plane to become unstable. Assuming that the initial settings are symmetrical, the following procedure ensures that they remain symmetrical, even if an exception is raised – either as a result of a failure of the physical system or because of a program error.²

```

procedure Wing_Settings ( -- relevant parameters) is
begin
    -- carry out the required setting
    -- of slats and flaps;
    -- exceptions may be raised
exception
    when others =>
        -- ensure the settings are symmetrical
        -- re-raise exception to indicate
        -- a slatless and flapless landing
        raise;
end Wing_Settings;

```

Ada allows an alternative mechanism for programming last wishes using **controlled types**. With these types, it is possible to define subprograms that are called (automatically) when objects of the type:

- are created – *initialize*;
- cease to exist – *finalize*;
- are assigned a new value – *adjust*.

To gain access to these features, the type must be derived from **Controlled**, a predefined type declared in the library package `Ada.Finalization`. The package defines procedures for `Initialize`, `Finalize` and `Adjust`. When a type is derived from **Controlled**, these procedures may be overridden.

To use controlled types to support last wishes requires a dummy controlled variable to be declared in the procedure. The finalization procedure can then be used to ensure a termination condition in the presence of exceptions. In the above example, it would ensure that the slats and flaps have a symmetrical setting.

Suppressing exceptions

There is an aphorism which has become popular with programmers; it normally takes the form: ‘there is no such thing as a free lunch!’ One of the requirements for exception-handling facilities was that they should not incur run-time overheads unless exceptions were raised (R3). The facilities provided by Ada have been described, and on the surface they appear to meet this requirement. However, there will always be some overhead associated with detecting possible error conditions.

²This, of course, is a crude example used to illustrate an approach; it is not necessarily the approach that would be taken in practice.

For example, Ada provides a standard exception called `Constraint_Error` that is raised when a null pointer is used, or where there is an array bound error, or where an object is assigned a value outside its permissible range. In order to catch these error conditions, a compiler must generate appropriate code. For instance, when an object is being accessed through a pointer, a compiler will, in the absence of any global flow control analysis (or hardware support), insert code which tests to see if the pointer is null before accessing the object. Although hidden from the programmer, this code will be executed even when no exception is to be raised. If a program uses many pointers, this can result in a significant overhead both in terms of execution time and code size. Furthermore, the presence of the code may require it to be tested during any validation process, and this may be difficult to do.

The Ada language does recognize that the standard exceptions raised by the run-time environment may be too costly for a particular application. Consequently, it provides a facility by which these checks can be suppressed. This is achieved by use of the `Suppress` pragma which eliminates a whole range of run-time checks. The pragma affects only the compilation unit in which it appears. Of course, if a run-time error check is suppressed and subsequently the error occurs, then the language considers the program to be 'erroneous' and the subsequent behaviour of the program is undefined.

A full example

The following package illustrates the use of exceptions in an abstract data type which implements a single `Stack`. This example was chosen as it enables the full specification and body to be given without leaving anything to the reader's imagination.

The package is generic and, therefore, can be instantiated for different types.

```
generic
  Size : Natural := 100;
  type Item is private;
package Stack is

  Stack_Full, Stack_Empty : exception;

  procedure Push(X:in Item);
  procedure Pop(X:out Item);

end Stack;

package body Stack is
  type Stack_Index is new Integer range 0 .. Size-1;
  type Stack_Array is array(Stack_Index) of Item;
  type Stack is
    record
      S : Stack_Array;
      Sp : Stack_Index := 0;
    end record;
  Stk : Stack;

  procedure Push(X:in Item) is
  begin
    if Stk.Sp = Stack_Index'Last then
```



```

    raise Stack_Full;
end if;
Stk.Sp := Stk.Sp + 1;
Stk.S(Stk.Sp) := X;
end Push;

procedure Pop(X:out Item) is
begin
    if Stk.Sp = Stack_Index'First then
        raise Stack_Empty;
    end if;
    X := Stk.S(Stk.Sp);
    Stk.Sp := Stk.Sp - 1;
end Pop;
end Stack;

```

It may be used as follows:

```

with Stack;
with Text_Io;
procedure Use_Stack is
    package Integer_Stack is new Stack(Item => Integer);
    X : Integer;
    use Integer_Stack;
begin
    ...
    Push(X);
    ...
    Pop(X);
    ...
exception
    when Stack_Full =>
        Text_Io.Put_Line("stack overflow!");
    when Stack_Empty =>
        Text_Io.Put_Line("stack empty!");
end Use_Stack;

```

Difficulties with the Ada model of exceptions

Although the Ada language provides a comprehensive set of facilities for exception handling, there are some difficulties with its ease of use.

- (1) **Exceptions and packages** – exceptions that can be raised by the use of a package are declared in the package specification along with any subprograms that can be called. Unfortunately, it is not obvious which subprograms can raise which exceptions. If the users of the package are unaware of its implementation, they must attempt to associate the names of exceptions with the subprogram names. In the stack example given above, the user could assume that the exception `Stack_Full` is raised by the procedure `Pop` and not `Push`! For large packages, it may not be obvious which exceptions can be raised by which subprograms. The programmer in this case must resort to either enumerating all possible exceptions every time a subprogram is called, or to the use of **when others**. Writers of packages should therefore indicate which subprograms can raise which exceptions using comments.

- (2) **Parameter passing** – Ada does not allow a full range of parameters to be passed to handlers only a character string. This can be inconvenient if an object of a particular type needs to be passed.
- (3) **Scope and propagation** – it is possible for exceptions to be propagated outside the scope of their declaration. Such exceptions can only be trapped by **when others**. However, they may go back into scope again when propagated further up the dynamic chain. This is disconcerting, although probably inevitable when using a block structured language and exception propagation.

3.3.2 Java

Java is similar to Ada in that it supports a termination model of exception handling. However, Java's exceptions, unlike Ada, are integrated into the object-oriented model.

Exception declaration

In Java, all exceptions are subclasses of the predefined class `java.lang.Throwable`. The language also defines other classes, for example: `Error`, `Exception`, and `RuntimeException`. The relationship between these is depicted in Figure 3.3. Throughout this book, the term Java exception is used to denote any class derived from `Throwable` not just those derived from `Exception`.

Objects derived from `Error` describe internal errors and resource exhaustion in the Java run-time support system. Although these errors clearly have a major impact on the program, there is little that the program can do when they are thrown (raised) as no assumptions can be made concerning the integrity of the system.

Objects derived from the `Exception` hierarchy represent errors that programs can handle and potentially throw themselves. `RuntimeException`s are those

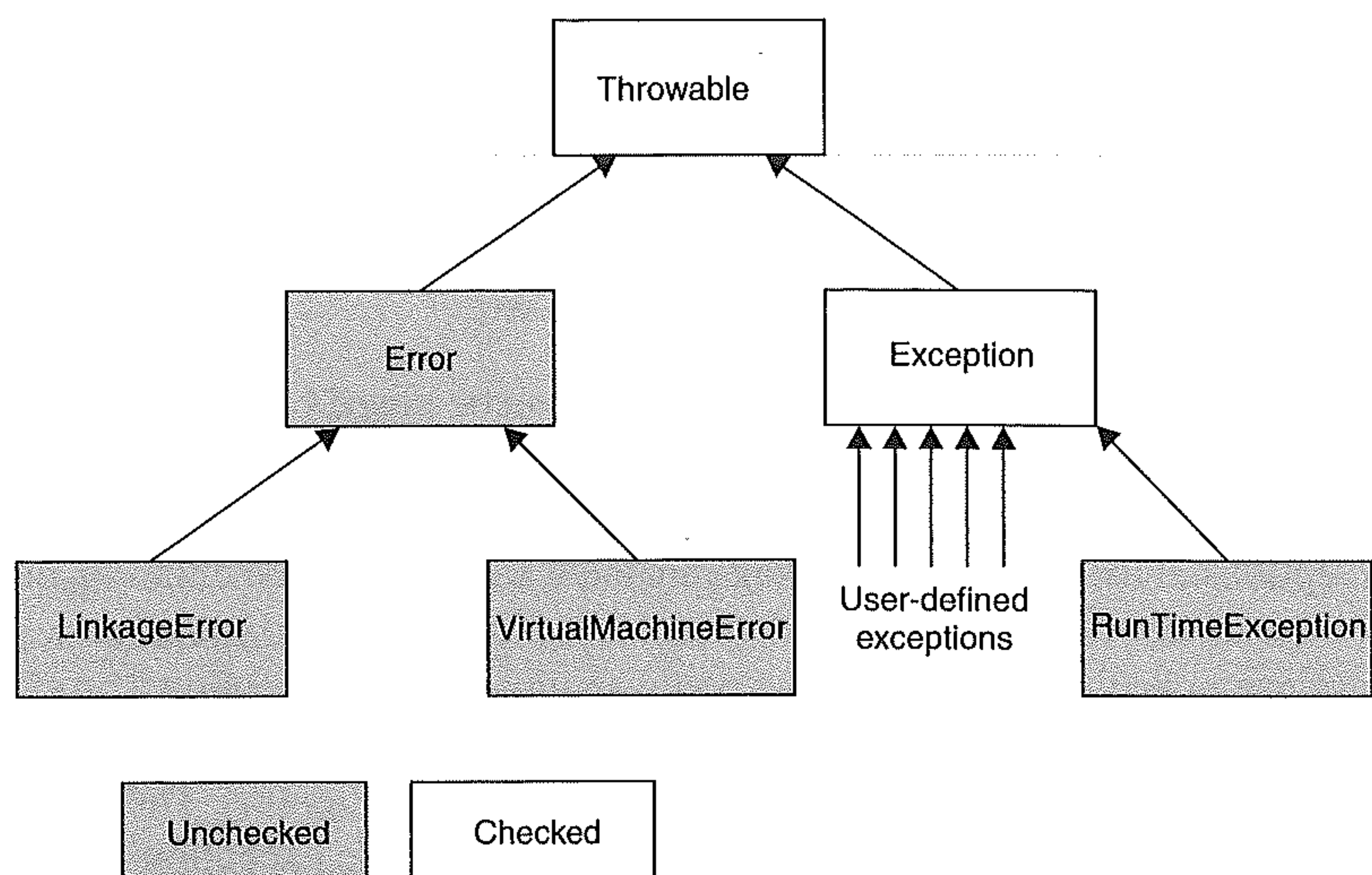


Figure 3.3 The Java predefined `Throwable` class hierarchy.

exceptions that are raised by the run-time support system as a result of a program error. They include errors such as those resulting from a bad cast `ClassCastException`, array bounds error (`IndexOutOfBoundsException`), a null pointer access (`NullPointerException`), integer divide by zero (`ArithmeticException`), etc.

Throwable objects which are derived from `Error` or `RuntimeExceptions` are called **unchecked** exceptions. This means that the Java compiler will not expect them to be identified in the `throws` clause of a method's declaration (see below).

Consider, for example, the temperature controller example given in Section 3.3.1. In Java, this might be written as follows. First a class has to be declared to represent a constraint error for integer subtypes. As this is a general error condition, it may have already been provided in a library. The private variables of the class will contain information about the cause of the error.

```
public class IntegerConstraintError extends Exception {

    private int lowerRange, upperRange, value;

    public IntegerConstraintError(int L, int U, int V) {
        super(); // call constructor on parent class
        lowerRange = L;
        upperRange = U;
        value = V;
    }

    public String getMessage() {
        return ("Integer Constraint Error: Lower Range " +
            java.lang.Integer.toString(lowerRange) + " Upper Range " +
            java.lang.Integer.toString(upperRange) + " Found " +
            java.lang.Integer.toString(value));
    }
}
```

Now the temperature type can be introduced.

```
import exceptionLibrary.IntegerConstraintError;
public class Temperature {
    private int T;

    public Temperature(int initial) throws IntegerConstraintError {
        // constructor
        ...;
    }
    public void setValue(int V) throws IntegerConstraintError {
        ...;
    }

    public int readValue() {
        ...;
        return T;
    }

    // both the constructor and setValue can throw an
    // IntegerConstraintError
}
```

In the above code, member functions which can throw the `IntegerConstraintError` exception are formally labelled. Contrast this with the Ada approach, which relies on comments.

Now the class `TemperatureController` can be defined. It declares a type to represent the failed actuator exception. In this case, no data is passed with the object.

```
class ActuatorDead extends Exception {
    public String getMessage() {
        return ("Actuator Dead");
    }
}

class TemperatureController {
    public TemperatureController(int T)
        throws IntegerConstraintError {
        currentTemperature = new Temperature(T);
    }

    Temperature currentTemperature;

    public void setTemperature(int T)
        throws ActuatorDead, IntegerConstraintError {
        currentTemperature.setValue(T);
    }

    int readTemperature() {
        return currentTemperature.readValue();
    }
}
```

In general, each function must specify a list of throwable checked exceptions **throws** A, B, C – in which case the function may throw any exception in this list and any of the unchecked exceptions. A, B and C must be subclasses of `Exception`. If a function attempts to throw an exception that is not allowed by its throws list, then a compilation error occurs.

Throwing an exception

Raising an exception in Java is called **throwing** the exception. For example, consider now the full implementation of the `Temperature` class outlined in the previous section.

```
import exceptionLibrary.IntegerConstraintError;

class Temperature {
    int T;

    void check(int value) throws IntegerConstraintError {
        if (value > 100 || value < 0) {
            throw new IntegerConstraintError(0, 100, value);
        }
    }
}
```



```

public Temperature(int initial) throws IntegerConstraintError {
    // constructor
    check(initial);
    T = initial;
}

public void setValue(int V) throws IntegerConstraintError {
    check(V);
    T = V;
}

public int readValue() {
    return T;
}
}

```

Here **throw new IntegerConstraintError(0, 100, value)** creates and throws an object of type (*IntegerConstraintError*) with the appropriate values for its instance variables.

Exception handling

An exception can only be handled in Java from within a **try-block**. Each handler is specified using a **catch** statement. Consider the following.

```

// given TemperatureController
try {
    TemperatureController TC = new TemperatureController(20);

    TC.setTemperature(100);
    // statements which manipulate the temperature
}
catch (IntegerConstraintError error) {
    // exception caught, print error message on
    // the standard output
    System.out.println(error.getMessage());
}
catch (ActuatorDead error) {
    System.out.println(error.getMessage());
}

```

The **catch** statement is like a function declaration, the parameter of which identifies the exception type to be caught. Inside the handler, the object name behaves like a local variable.

A handler with parameter type *T* will catch a thrown object of type *E* if:

- (1) *T* and *E* are the same type; or
- (2) *T* is a parent (super) class of *E* at the throw point.

It is this last point that makes the Java exception-handling facility very powerful. In the above example, two exceptions are derived from the *Exception* class; *IntegerConstraintError* and *ActuatorDead*. The following try-block will catch both exceptions.

```

try {
    // statements which might raise the exception
    // IntegerConstraintError or ActuatorDead
}
catch(Exception E) {
    // handler will catch all exceptions of type exception and
    // any derived type; but from within the handler only the
    // methods of Exception are accessible
}

```

Of course, a call to `E.getMessage` will dispatch to the appropriate routine for the type of object thrown. Indeed, the `catch(Exception E)` is equivalent to Ada's **when others**.

Exception propagation

As with Ada, if no exception is found in the calling context of a function, the calling context is terminated and a handler is sought in a try-block within its calling context. Hence, Java supports exception propagation.

Last wishes

Section 3.3.1 discussed how **when others** could be used to program last wishes in Ada. Clearly a `catch(Exception E)` can also be used to achieve this effect. However, Java also supports a **finally** clause as part of a try-block. The code attached to this clause is guaranteed to execute whatever happens in the try-block irrespective of whether exceptions are thrown, caught or propagated, or, indeed, even if there are no exceptions thrown at all.

```

try {
    ...
}
catch(...) { ... }
finally {
    // code that will be executed under all circumstances
}

```

The same effect can be achieved in Ada using controlled variables and finalization (see Section 3.3.1).

A full example

The following is the stack example in Java that was previously given for Ada. There are two points about this example which are of interest.

Firstly, the stack is generic in that any object can be passed as a parameter to push. Hence, there is no need to instantiate the stack (as in the Ada case). Furthermore, each stack can handle more than one object type (unlike the Ada case). Of course, this is because the Java stack is actually a stack of references to objects. It would be possible to explicitly program this in Ada.

The second point is that the Java example will only handle objects and not the primitive types like integer. This is in contrast to the Ada example, which will handle all types.


```

public class FullStackException extends Exception {
    public FullStackException() {
    }
}

public class EmptyStackException extends Exception {
    public EmptyStackException() {
    }
}

public class Stack {

    public Stack(int capacity) {
        stackArray = new Object[capacity];
        stackIndex = 0;
        stackCapacity = capacity;
    }

    public void push(Object item) throws FullStackException {
        if(stackIndex == stackCapacity) throw new FullStackException();
        stackArray[stackIndex++] = item;
    }

    public synchronized Object pop() throws EmptyStackException {
        if(stackIndex == 0) throw new EmptyStackException();
        return stackArray[--stackIndex];
    }

    protected Object stackArray[];
    protected int stackIndex;
    protected int stackCapacity;
}

```

The above classes can be used as follows:

```

public class UseStack {

    public static void main(...) {
        Stack S = new Stack(100);
        try {
            ...
            S.push(SomeObject);
            ...
            SomeObject = S.pop();
            ...
        }
        catch (FullStackException F) { ... }
        catch (EmptyStackException E) { ... }
    }
}

```

Finally, it should be noted that Java provides a standard Stack class in its util package.

3.3.3 C

C does not define any exception-handling facilities within the language. Such an omission clearly limits the usefulness of the language in the structured programming of reliable systems. However, it is possible to provide some form of exception-handling mechanism by using the macro facility of the language. To illustrate this approach, the implementation of a simple Ada-like exception-handling macros will be considered. The approach is based on that given by Lee (1983).

To implement a termination model in C, it is necessary to save the status of a program's registers, and so on, on entry to an exception domain and then restore them if an exception occurs. Traditionally C has been associated with Unix, and the POSIX facilities of `setjmp` and `longjmp` can be used for this purpose. The routine `setjmp` saves the program status and returns a 0; the routine `longjmp` restores the program status and results in the program abandoning its current execution and restarting from the position where `setjmp` was called. This time, however, `setjmp` returns the value passed by `longjmp`. The following code structure is needed:

```
/* begin exception domain */

typedef char *exception;
/* a pointer type to a character string */
exception error = "error";
/* the representation of an exception named "error" */

if((current_exception = (exception) setjmp(save_area)) == 0) {
    /* save the registers and so on in save_area */
    /* 0 is returned */

    /* the guarded region */

    /* when an exception "error" is identified */
    longjmp(save_area, (int) error);
    /* no return */
}
else {
    if(current_exception == error) {
        /* handler for "error" */
    }
    else {
        /* re-raise exception in surrounding domain */
    }
}
```

The above code is clearly difficult to understand. However, a set of macros can be defined to help structure the program.

```
#define NEW_EXCEPTION(name) ...
/* code for declaring an exception */
#define BEGIN ...
/* code for entering an exception domain */
#define EXCEPTION ...
/* code for beginning exception handlers */
#define END ...
```



```

    /* code for leaving an exception domain */
#define RAISE(name) ...
    /* code for raising an exception */
#define WHEN(name) ...
    /* code for handler */
#define OTHERS ...
    /* code for catch all exception handler */

```

Consider now the following example:

```

NEW_EXCEPTION(sensor_high);
NEW_EXCEPTION(sensor_low);
NEW_EXCEPTION(sensor_dead);
/* other declarations */

BEGIN
    /* statements which may cause the above exceptions */
    /* to be raised; for example */
    RAISE(sensor_high);

EXCEPTION
    WHEN(sensor_high)
        /* take some corrective action */
    WHEN(sensor_low)
        /* take some corrective action */
    WHEN(OTHERS)
        /* sound an alarm */
END;

```

The above provides a simple termination model similar to Ada's and Java's.

3.4 Recovery blocks and exceptions

In Chapter 2, the notion of recovery blocks was introduced as a mechanism for fault-tolerant programming. Its main advantage over forward error recovery mechanisms is that it can be used to recover from unanticipated errors, particularly from errors in the design of software components. So far in this chapter, only anticipated errors have been considered, although catch-all exception handlers can be used to trap unknown exceptions. In this section, the implementation of recovery blocks using exceptions and exception handlers is described.

As a reminder, the structure of a recovery block is shown below:

```

ensure <acceptance test>
by
    <primary module>
else by
    <alternative module>
else by
    <alternative module>
    .
    .
    .
else by
    <alternative module>
else error

```

The error detection facility is provided by the acceptance test. This test is simply the negation of a test which would raise an exception using forward error recovery. The only problem is the implementation of state saving and state restoration. In the example below, this is shown as an Ada package which implements a recovery cache. The procedure `Save` stores the state of the global and local variables of the program in the recovery cache; this does not include the values of the program counter, stack pointer and so on. A call of `Restore` will reset the program variables to the states saved.

```
package Recovery_Cache is
  procedure Save;
  procedure Restore;
end Recovery_Cache;
```

Clearly, there is some magic going on inside the package which will require support from the run-time system and possibly even hardware support for the recovery cache. Also, this may not be the most efficient way to perform state restoration. It may be more desirable to provide more basic primitives, and to allow the program to use its knowledge of the application to optimize the amount of information saved (Rogers and Wellings, 2000).

The purpose of the next example is to show that given recovery cache implementation techniques, recovery blocks can be used in an exception-handling environment. Notice also that by using exception handlers, forward error recovery can be achieved before restoring the state. This overcomes a criticism of recovery blocks: that it is difficult to reset the environment.

The recovery block scheme can, therefore, be implemented using a language with exceptions plus a bit of help from the underlying run-time support system. For example, in Ada the structure for a triple redundant recovery block would be:

```
procedure Recovery_Block is
  Primary_Failure, Secondary_Failure,
    Tertiary_Failure: exception;
  Recovery_Block_Failure : exception;
  type Module is (Primary, Secondary, Tertiary);
  function Acceptance_Test return Boolean is
  begin
    -- code for the acceptance test
  end Acceptance_Test;

  procedure Primary is
  begin
    -- code for primary algorithm
    if not Acceptance_Test then
      raise Primary_Failure;
    end if;
  exception
    when Primary_Failure =>
      -- forward recovery to return environment
      -- to the required state
      raise;
    when others =>
      -- unexpected error
      -- forward recovery to return environment
```



```

    -- to the required state
    raise Primary_Failure;
end Primary;

```

```

procedure Secondary is
begin
    -- code for secondary algorithm
    if not Acceptance_Test then
        raise Secondary_Failure;
    end if;
exception
    when Secondary_Failure =>
        -- forward recovery to return environment
        -- to the required state
        raise;
    when others =>
        -- unexpected error
        -- forward recovery to return environment
        -- to the required state
        raise Secondary_Failure;
end Secondary;

```

```

procedure Tertiary is
begin
    -- code for tertiary algorithm
    if not Acceptance_Test then
        raise Tertiary_Failure;
    end if;
exception
    when Tertiary_Failure =>
        -- forward recovery to return environment
        -- to the required state
        raise;
    when others =>
        -- unexpected error
        -- forward recovery to return environment
        -- to the required state
        raise Tertiary_Failure;
end Tertiary;

```

```

begin
    Recovery_Cache.Save;
    for Try in Module loop
        begin
            case Try is
                when Primary => Primary; exit;
                when Secondary => Secondary; exit;
                when Tertiary => Tertiary;
            end case;
            exception
                when Primary_Failure =>
                    Recovery_Cache.Restore;
                when Secondary_Failure =>
                    Recovery_Cache.Restore;
                when Tertiary_Failure =>
                    Recovery_Cache.Restore;

```

```
        raise Recovery_Block_Failure;
    when others =>
        Recovery_Cache.Restore;
        raise Recovery_Block_Failure;
    end;
end loop;
end Recovery_Block;
```

Summary

This chapter has studied the various models of exception handling for sequential processes. Although many different models exist they all address the following issues.

- **Exception representation** – an exception may, or may not, be explicitly represented in a language.
- **The domain of an exception handler** – associated with each handler is a domain which specifies the region of computation during which, if an exception occurs, the handler will be activated. The domain is normally associated with a block, subprogram or a statement.
- **Exception propagation** – this is closely related to the idea of an exception domain. It is possible that when an exception is raised there is no exception handler in the enclosing domain. In this case, either the exception can be propagated to the next outer level enclosing domain, or it can be considered to be a programmer error (which can often be flagged at compilation time).
- **Resumption or termination model** – this determines the action to be taken after an exception has been handled. With the resumption model, the invoker of the exception is resumed at the statement after the one at which the exception was invoked. With the termination model, the block or procedure containing the handler is terminated, and control is passed to the calling block or procedure. The hybrid model enables the handler to choose whether to resume or to terminate.
- **Parameter passing to the handler** – this may or may not be allowed.

The exception-handling facilities of various languages are summarized in Table 3.1.

Language	Domain	Propagation	Model	Parameters
Ada	Block	Yes	Termination	Limited
Java	Block	Yes	Termination	Yes
C++	Block	Yes	Termination	Yes
CHILL	Statement	No	Termination	No
CLU	Statement	No	Termination	Yes
Mesa	Block	Yes	Hybrid	Yes

Table 3.1 The exception-handling facilities of various languages.

It is not unanimously accepted that exception-handling facilities should be provided in a language. To sceptics, an exception is a `goto` where the destination is undeterminable and the source is unknown. They can, therefore, be considered to be the antithesis of structured programming. This, however, is not the view taken in this book.

Further reading

- Buhr, P. A. and Mok, W. Y. R (2000) Advanced exception handling mechanisms, *IEEE Transactions on Software Engineering*, **26**(9), 820–836.
- Cristian, F. (1982) Exception handling and software fault tolerance, *IEEE Transactions on Computing*, **c-31**(6), 531–540.
- Cui, Q. and Gannon, J. (1992) Data-oriented exception handling, *IEEE Transactions on Software Engineering*, **18**(5), 393–401.
- Lee, P. A. (1983) Exception handling in C programs, *Software – Practice and Experience*, **13**(5), 389–406.
- Papurt, D. M. (1998) The use of exceptions, *JOOP*, **11**(2), 13–17.

Exercises

- 3.1 Compare and contrast the exception-handling and recovery block approaches to software fault tolerance.
- 3.2 Give examples of:
 - (1) a synchronous exception detected by the application
 - (2) an asynchronous exception detected by the application
 - (3) a synchronous exception detected by the execution environment
 - (4) an asynchronous exception detected by the execution environment
- 3.3 The package `Character_Io` whose specification is given below, provides a function for reading characters from the terminal. It also provides a procedure for throwing away all the remaining characters on the current line. The package may raise the exception `Io_Error`.

```
package Character_Io is
  function Get return Character;
    -- reads a character from the terminal

  procedure Flush;
    -- throw away all character on the current line

  Io_Error : exception;
end Character_Io;
```

Another package `Look` contains the function `Read` which scans the current input line looking for the punctuation characters comma (,) period (.) and semicolon (;). The function will return the next punctuation character found or raise the exception `Illegal_Punctuation` if a non-alphanumeric character is found. If,

during the process of scanning the input line, an `Io_Error` is encountered, then the exception is propagated to the caller of `Read`. The specification of `Look` is given below.

```
with Character_Io; use Character_Io;
package Look is
  type Punctuation is (Comma, Period, Semicolon);
  function Read return Punctuation;
    -- reads the next , . or ; from the terminal

  Illegal_Punctuation : exception;
end Look;
```

Sketch the package body of `Look` using the `Character_Io` package for reading characters from the terminal. On receipt of a legal punctuation character, an illegal punctuation character or an `Io_Error` exception, the remainder of the input line should be discarded. You may assume that an input line will always have a legal or illegal punctuation character and that `Io_Errors` occur at random. Using the `Look` package, sketch the code of a procedure `Get_Punctuation` which will always return the next punctuation character in spite of the exceptions that `Look` raises. An infinite input stream may be assumed.

- 3.4 In a process control application, gas is heated in an enclosed chamber. The chamber is surrounded by a coolant which reduces the temperature of the gas by conduction. There is also a valve which when open releases the gas into the atmosphere. The operation of the process is controlled by an Ada package whose specification is given below. For safety reasons, the package recognizes several error conditions; these are brought to the notice of the user of the package by the raising of exceptions. The exception `Heater_Stuck_On` is raised by the procedure `Heater_Off` when it is unable to turn the heater off. The exception `Temperature_Still_Rising` is raised by the `Increase_Coolant` procedure if it is unable to decrease the temperature of the gas by increasing the flow of the coolant. Finally, the exception `Valve_Stuck` is raised by the `Open_Valve` procedure if it is unable to release the gas into the atmosphere.

```
package Temperature_Control is
  Heater_Stuck_On, Temperature_Still_Rising,
  Valve_Stuck : exception;

  procedure Heater_On;
    -- turn on heater

  procedure Heater_Off;
    -- turn off heater
    -- raises Heater_Stuck_On

  procedure Increase_Coolant;
    -- Causes the flow of coolant which surrounds the
    -- chamber to increase until the temperature reaches
    -- a safe level raises Temperature_Still_Rising
```



```

procedure Open_Valve;
  -- opens a valve to release some of the gas thereby
  -- avoiding an explosion
  -- raises Valve_Stuck

procedure Panic;
  -- sounds an alarm and calls the fire,
  -- hospital and police services
end Temperature_Control;

```

Write an Ada procedure which when called will attempt to turn off the heater in the gas chamber. If the heater is stuck on then the flow of coolant surrounding the chamber should be increased. If the temperature still rises then the escape valve should be opened to release the gas. If this fails then the alarm must be sounded and the emergency services informed.

- 3.5 Write a general-purpose generic Ada package to implement nested recovery blocks. (Hint – the primary, secondary modules, acceptance test and so on, should be encapsulated in a package and passed as a parameter to the generic.)
- 3.6 To what extent could the Ada exception-handling facilities be implemented outside the language by a standard package?
- 3.7 How would you defend the statement that an Ada exception is a `goto` where the destination is undeterminable and the source is unknown? Would the same argument hold for (a) the resumption model of exception handling and (b) the CHILL termination model?
- 3.8 Compare the exception domains of the following apparently identical pieces of code (the variable `Initial` is of type integer).

```

procedure Do_Something is
  subtype Small_Int is Integer range -16..15;
  A : Small_Int := Initial;
begin
  ...
end Do_Something;

```

```

procedure Do_Something is
  subtype Small_Int is Integer range -16..15;
  A : Small_Int;
begin
  A := Initial;
  ...
end Do_Something;

```

```

procedure Do_Something is
  subtype Small_Int is Integer range -16..15;
  A : Small_Int;
begin
  begin
    A := Initial;
    ...
  end;
end Do_Something;

```

- 3.9** Show how the C macros given in Section 3.3.3 can be implemented.
- 3.10** Consider the following program:

```

I : Integer := 1;
J : Integer := 0;
K : Integer := 3;

procedure Primary is
begin
    J := 20;
    I := K*J/2
end Primary;

procedure Secondary is
begin
    I := J;
    K := 4;
end Secondary;

procedure Tertiary is
begin
    J := 20;
    I := K*J/2
end Tertiary;

```

This code is to be executed in a recovery block environment and in an exception-handling environment. The following is the recovery block environment:

```

ensure I = 20
by
    Primary;
else by
    Secondary;
else by
    Tertiary;
else
    I := 20;
end;

```

The following is the exception-handling environment:

```

Failed : exception;
type Module is (P,S,T);

for Try in Module loop
    begin
        case Try is
            when P =>
                Primary;
                if I /= 20 then
                    raise Failed;
                end if;
                exit;
            when S =>

```



```

    Secondary;
    if I /= 20 then
        raise Failed;
    end if;
    exit;
when T =>
    Tertiary;
    if I /= 20 then
        raise Failed;
    end if;
    exit;
end case;
exception
    when Failed =>
        if Try = T then
            I := 20;
        end if;
end;
end loop;

```

Assuming the *termination model* of exception handling, compare and contrast the execution of *both* the recovery block environment and the exception-handling environment code fragments.

- 3.11 Illustrate how recovery blocks can be implemented using C++ and Java. Hint: see Rubira-Calsavara and Stroud (1994).
- 3.12 Redo the answer to Exercise 3.3 using Java.
- 3.13 Redo the answer to Exercise 3.4 using Java.
- 3.14 Explain under what circumstances **when others** (*e:Exception_Id*) in Ada is not equivalent to **catch** (*Exception e*) in Java.
- 3.15 What are the advantages and disadvantages of having exceptions integrated into the OOP model?
- 3.16 A subset of Ada has been defined for use in safety-critical embedded systems. The language has formal semantics and a set of tools that allow static analysis techniques to be performed. It has no exception-handling facility. The designers argue that exceptions cannot occur if the program has been proved correct. What arguments can be put forward for the introduction of exception-handling facilities into the language?

Chapter 4

Concurrent programming

4.1	Processes and tasks/threads	4.7	Multiprocessor and distributed systems
4.2	Concurrent execution	4.8	A simple embedded system
4.3	Task representation	4.9	Language-supported versus operating-system-supported concurrency
4.4	Concurrent execution in Ada		Summary
4.5	Concurrent execution in Java		Further reading
4.6	Concurrent execution in C/Real-Time POSIX		Exercises

Concurrent programming is the name given to the programming notations and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems.

There are three main motivations for wanting to write concurrent programs.

- (1) To model parallelism in the real world – real-time and embedded programs have to control and interface with real-world entities (robots, conveyor belts, etc.) that are inherently parallel. Reflecting the parallel nature of the system in the structures of the program makes for a more readable, maintainable and reliable application.
- (2) To fully utilize the processor – modern processors run at speeds far in excess of the input and output devices with which they must interact. A sequential program that is waiting for I/O is unable to perform any other operation.
- (3) To allow more than one processor to solve a problem – a sequential program can only be executed by one processor (unless the compiler has transformed the program into a concurrent one). Modern hardware platforms consist of multiple processors to obtain more powerful execution environments. A concurrent program is able to exploit this true parallelism and obtain faster execution.

From a concurrency perspective, the implementation platform can be considered irrelevant. However, in some application areas the amount or presence of parallelism is critical to the success of the program. For example, in

high-performance computing there is a requirement to get maximum performance from the platform. Inevitably, creating many tasks adds overhead. Hence, there may be some advantages from limiting the number of concurrent activities to the numbers of CPUs, or rather writing the application so that the number of activities created at run-time matches the number of processors available.

From a theoretical point of view, Amdahl's law gives the relationship between the expected speedup of parallelizing implementations of an algorithm. If P is the proportion of that algorithm's code that can benefit from parallelization and N is the number of processors, the maximum speedup that can be achieved by using these N processors is

$$\frac{1}{(1 - P) + \frac{P}{N}} \quad (4.1)$$

In the limit, as N tends to infinity, the maximum speedup tends to $1/(1 - P)$. Consequently, if only 50% of the algorithm is amenable to parallel execution, then the maximum possible speedup is only 2. Consequently, there is no point (for efficiency considerations) in having 100 processors to perform the algorithm.

Of course, speedup is only one reason for the use of concurrent activities. Hence, even though the number of processors might be limited, there are still advantages to having more concurrent activities than processors. This, and the following two chapters, concentrate on the issues associated with general concurrent programming.

4.1 Processes and tasks/threads

Any language, natural or computer, has the dual property of enabling expression while at the same time limiting the framework within which that expressive power may be applied. If a language does not support a particular notion or concept then those that use the language cannot apply that notion and may even be totally unaware of its existence.

Pascal, C, FORTRAN and COBOL share the common property of being sequential programming languages. Programs written in these languages have a single thread of control. They start executing in some state and then proceed, by executing one statement at a time, until the program terminates. The path through the program may differ due to variations in input data, but for any particular execution of the program there is only one path. This is not adequate for the programming of real-time systems.

Following the pioneering work of Dijkstra (1968), a concurrent program is conventionally viewed as consisting of a collection of autonomous sequential processes, executing (logically) in parallel. Concurrent programming languages all incorporate, either explicitly or implicitly, the notion of process; each process itself has a single thread of control. All operating systems provide facilities for creating concurrent processes. Usually, each process executes in its own virtual machine to avoid interference from other, unrelated, processes. Each process is, in effect, a single program. However, in recent years there has been a tendency to provide the facilities for processes to be created within programs.

Modern operating systems allow processes created within the same program to have unrestricted access to shared memory (such processes are called **threads** or **tasks**). Hence, in operating systems like those conforming to the POSIX API, it is necessary to distinguish the concurrency between programs (processes) from the concurrency within a program (threads/tasks). Often, there is also a distinction between threads which are visible to the operating system (often called *kernel-level* threads) and those that are supported solely by library routines (*library-level* threads). For example, Windows XP/2000 supports threads and **fibers**, the latter being invisible to the kernel. Hybrid models are also possible where a process can be allocated a maximum number of kernel-level threads and a thread library (or sometimes the kernel itself) is responsible for mapping the process's threads to the appropriate library or kernel-level threads.

*In this book the term **task** or **thread** will be used (interchangeably) to represent a single thread of control. The term process will be used to indicate one or more threads/tasks executing within its own shared memory context. With most concurrent programming languages, it is the thread/task abstraction that is supported rather than the process abstraction.*

Where a concurrent program is being executed on top of an operating system, its Run-Time Support System (RTSS) can either map the program's notion of a task onto the underlying operating system's notion of a thread, or it can make the program's notion of task invisible to the operating system (and in effect implement its own fibers library). Of course, in the latter case, the RTSS must ensure that all input/output operations are asynchronous, otherwise the whole concurrent program will block every time one of its tasks performs an operation requiring operating system support.

The actual implementation (that is, execution) of a collection of tasks usually takes one of three forms. Tasks can either:

- (1) multiplex their executions on a single processor;
- (2) multiplex their executions on a multiprocessor system where all processors have access to common shared memory (for example, a symmetric multiprocessor (SMP) system);
- (3) multiplex their executions on several processors where there is no common shared memory (for example, a distributed system).

Hybrids of these three methods are also possible, for example, Non-Uniform Memory Architectures (NUMA) where there may be a mixture of shared and non-shared memory.

Only in cases (2) and (3) is there the possibility of true parallel execution of more than one task. The term **concurrent** indicates potential parallelism. Concurrent programming languages thus enable the programmer to express logically parallel activities without regard to their implementation.

The life of a task is illustrated, simply, in Figure 4.1. A task is created, moves into the state of initialization, proceeds to execution and termination. Note that some tasks may never terminate and that others, which fail during initialization, pass directly to termination without ever executing. After termination, a task goes to non-existing when it can no longer be accessed (as it has gone out of scope). Clearly, the most important state for a task is executing; however, as processors are limited not all tasks can be executing

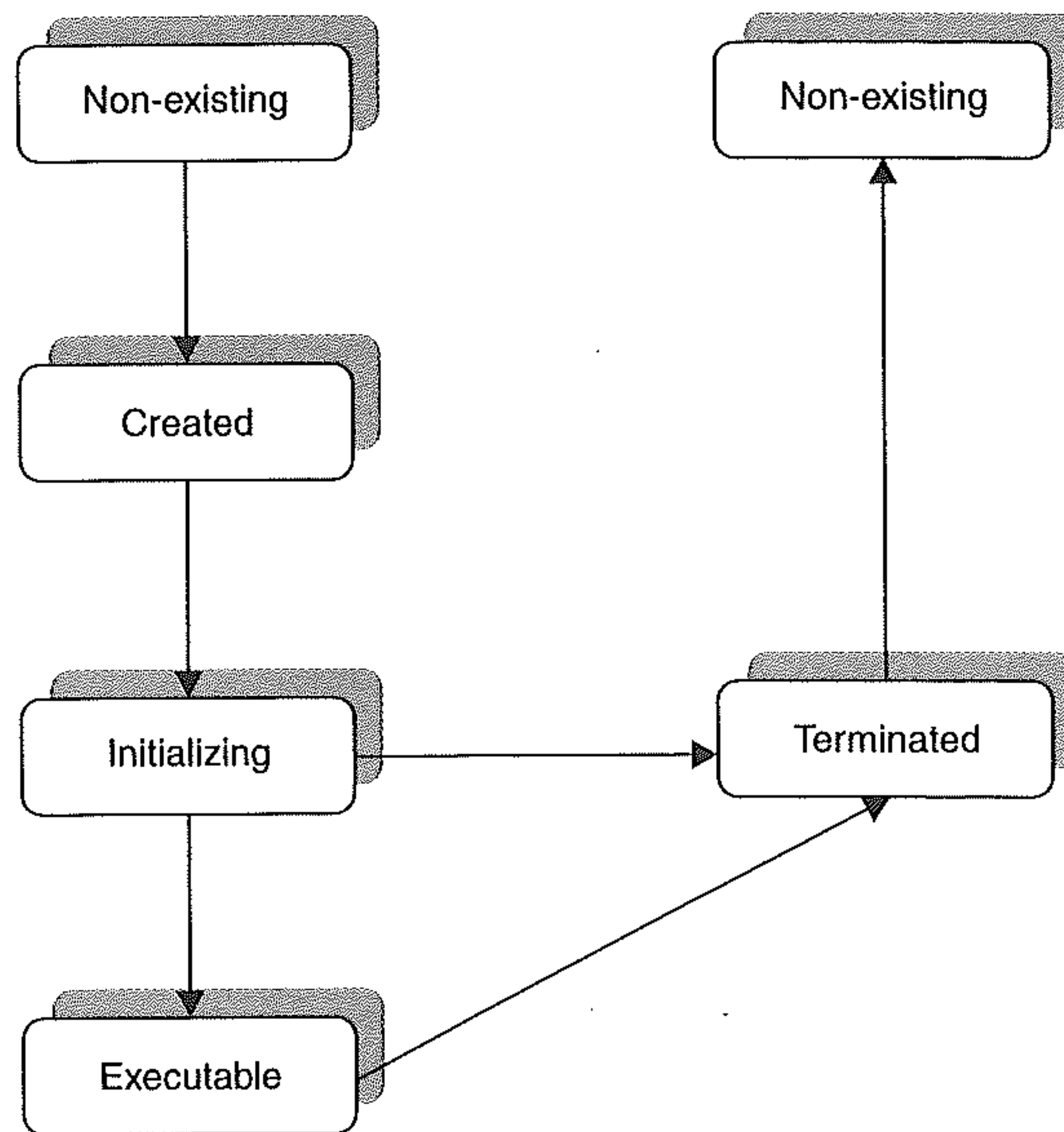


Figure 4.1 Simple state diagram for a task.

at once. Consequently, the term **executable** is used to indicate that the task could execute if there is a processor available.

From this consideration of a task, it is clear that the execution of a concurrent program is not as straightforward as the execution of a sequential program. Tasks must be created and terminated, and dispatched to and from the available processors. These activities are undertaken by the RTSS or Run-Time Kernel. The RTSS sits logically between the hardware (along with any provided operating system) and the application software. In reality, it may take one of a number of forms.

- (1) A software structure programmed as part of the application (that is, as one component of the concurrent program). C and C++ programs often provide their own thread libraries that are tailored to their applications' requirements.
- (2) A standard software system generated with the program object code by the compiler. This is normally the structure with Ada and Java programs.
- (3) A hardware structure microcoded into the processor for efficiency. For example, the aJile System's aJ100 processor directly supports the execution of Java byte code.

The algorithm used for scheduling by an RTSS (or an operating system) to decide which task to execute next if there is more than one executable will affect the time behaviour of the program. However, for well-constructed programs, the logical behaviour of a program will not be dependent on the scheduling. From the program's point of view, the RTSS (or operating system) is assumed to schedule tasks non-deterministically. For real-time systems, the characteristics of the scheduling are significant. They are considered further in Chapter 11.

4.1.1 Concurrent programming constructs

Although constructs for concurrent programming vary from one language (and operating system) to another, there are three fundamental facilities that must be provided. These allow the following:

- (1) the expression of concurrent activities (threads, tasks or processes);
- (2) the provision of synchronization mechanisms between concurrent activities;
- (3) primitives that support of communication between concurrent activities.

In considering the interaction of tasks, it is useful to distinguish between three types of behaviour:

- independent
- cooperating
- competing.

Independent tasks do not communicate or synchronize with each other. Cooperating tasks, by comparison, regularly communicate and synchronize their activities in order to perform some common operation. For example, a component of an embedded computer system may have several tasks involved in keeping the temperature and humidity of a gas in a vessel within certain defined limits. This may require frequent interactions.

A computer system has a finite number of resources which may be shared between processes and tasks; for example, peripheral devices, memory and processor power. In order for tasks to obtain their fair share of these resources, they must compete with each other. The act of resource allocation inevitably requires communication and synchronization between the tasks in the system. However, although these tasks communicate and synchronize in order to obtain resources, they are, essentially, independent.

Discussion of the facilities which support task creation and task interaction is the focus of this and the next two chapters.

4.2 Concurrent execution

Although the notion of tasks or threads is common to all concurrent programming languages, there are considerable variations in the models of concurrency adopted. These variations appertain to:

- structure
- level
- granularity
- initialization
- termination
- representation.

Language	Structure	Level
Concurrent Pascal	static	flat
occam2	static	nested
Modula-1	dynamic	flat
C/POSIX	dynamic	flat
Ada	dynamic	nested
Java	dynamic	nested
C#	dynamic	nested

Table 4.1 The structure and level characteristics of a number of concurrent programming languages.

The *structure* of a task may be classified as follows.

- **Static** – the number of tasks is fixed and known at compile-time.
- **Dynamic** – tasks are created at any time. The number of extant tasks is determined only at run-time.

Another distinction between languages comes from the *level* of parallelism supported. Again, two distinct cases can be identified.

- (1) **Nested** – tasks are defined at any level of the program text; in particular, tasks are allowed to be defined within other tasks.
- (2) **Flat** – tasks are defined only at the outermost level of the program text.

Table 4.1 gives the structure and level characteristics for a number of concurrent programming languages. In this table, the language C is considered to incorporate the Real-Time POSIX pthread mechanisms.

Within languages that support nested constructs, there is also an interesting distinction between what may be called **coarse** and **fine grain** parallelism. A coarse grain concurrent program contains relatively few tasks, each with a significant life history. By comparison, programs with a fine grain of parallelism will have a large number of simple tasks, some of which will exist for only a single action. Most concurrent programming languages, typified by Ada, display coarse grain parallelism. Occam2 is a good example of a concurrent language with fine grain parallelism.

When a task is created, it may need to be supplied with information pertinent to its execution (much as a procedure may need to be supplied with information when it is called). There are two ways of performing this **initialization**. The first is to pass the information in the form of parameters to the task; the second is to communicate explicitly with the task after it has commenced its execution. Most modern concurrent languages allow parameters to be passed at task creation time.

Task **termination** can be accomplished in a variety of ways. The circumstances under which tasks are allowed to terminate can be summarized as follows:

- (1) completion of execution of the task’s body;
- (2) suicide, by execution of a ‘self-terminate’ statement;

- (3) abortion, through the explicit action of another task;
- (4) occurrence of an untrapped error condition;
- (5) never: tasks are assumed to execute non-terminating loops;
- (6) when no longer needed.

With nested levels, hierarchies of tasks can be created and intertask relationships formed. For any task, it is useful to distinguish between the task (or block) that is responsible for its creation and the task (or block) which is affected by its termination. The former relationship is known as **parent/child** and has the attribute that the parent may be delayed while the child is being created and initialized. The latter relationship is termed **guardian/dependant**. A task may be dependent on the guardian task itself or on an inner block of the guardian. The guardian is not allowed to exit from a block until all dependent tasks of that block have terminated (that is, a task cannot exist outside its scope). It follows that a guardian cannot terminate until all its dependants have also terminated. This rule has the particular consequence that a program itself will not be able to terminate until all tasks created within it have also terminated.

In some situations, the parent of a task will also be its guardian. This will be the case when using languages which allow only static task structures. With dynamic task structures (that are also nested), the parent and guardian may or may not be identical. This will be illustrated later in the discussion of Ada. Figure 4.2 includes the new states that have been introduced in the above discussion.

One of the ways a task may terminate (point (3) in the above list) is by the application of an abort statement. The existence of abort in a concurrent programming language is a question of some contention and is considered in Chapter 8 within the context of resource control. For a hierarchy of tasks, it is usually necessary for the abort of a guardian to imply the abort of all dependants (and their dependants and so on).

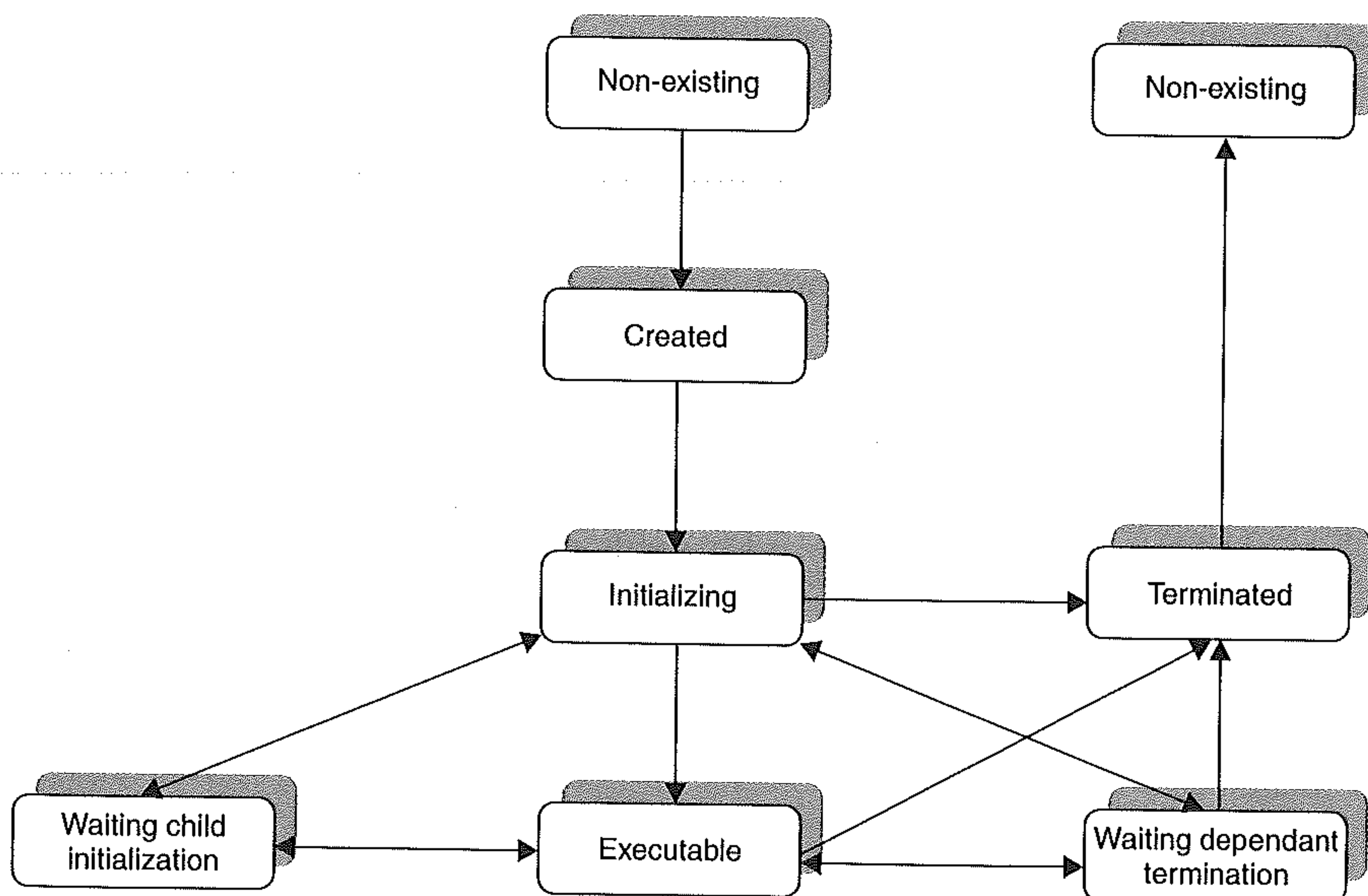


Figure 4.2 State diagram for a task.

The final circumstance for termination, in the above list, is considered in more detail when task communication methods are described. In essence, it allows a server task to terminate if all other tasks that could communicate with it have already terminated.

4.2.1 Tasks and objects

The object-oriented programming (OOP) paradigm encourages system (and program) builders to consider the artifact under construction as a collection of cooperating objects (or, to use a more neutral term, *entities*). Within this paradigm, it is constructive to consider two kinds of object – active and reactive. **Active** objects undertake spontaneous actions (with the help of a processor): they enable the computation to proceed. **Reactive** objects, by comparison, only perform actions when ‘invoked’ by an active object. Other programming paradigms, such as data-flow or real-time networks, also identify active agents and passive data.

Only active entities give rise to spontaneous actions. Resources are reactive but can control access to their internal states (and any real resources they control). Some resources can only be used by one agent at a time; in other cases the operations that can be carried out at a given time depend on the resources’ current states. A common example of the latter is a data buffer whose elements cannot be extracted if it is empty. The term **passive** will be used to indicate a reactive entity that can allow open access.

The implementation of resource entities requires some form of control agent. If the control agent is itself passive (such as a semaphore), then the resource is said to be **protected** (or **synchronized**). Alternatively, if an active agent is required to program the correct level of control, then the resource is in some sense active. The term **server** will be used to identify this type of entity, and the term **protected resource** to indicate the passive kind. These, together with **active** and **passive**, are the four abstract program entities used in this book.

In a concurrent programming language, active entities are represented by tasks/threads. Passive entities can be represented either directly as data variables or they can be encapsulated by some module/package/class construct that provides a procedural interface. Protected resources may also be encapsulated in a module-like construct and require the availability of a low-level synchronization facility. Servers, because they need to program the control agent, require a task.

From an object-oriented programming perspective:

- **passive object** – a reactive entity with no synchronization constraints, it needs an external thread of control for its methods to be executed;
- **protected object** – a reactive entity with synchronization constraints, it is typically shared between many threads and it needs an external thread of control for its methods to be executed;
- **active object** – an object with an explicit or implicit internal thread;
- **server object** – an active object with synchronization constraints, it is typically shared between many threads.

A key question for language designers is whether to support primitives for both protected resources and servers. Resources, because they typically use a low-level

control agent (for example, a semaphore), are normally implemented efficiently (at least on single-processor systems). However, they can be inflexible and lead to poor program structures for some classes of problems (this is discussed further in Chapter 5). Servers, because the control agent is programmed using a task, are eminently flexible. The drawback of this approach is that it can lead to a proliferation of tasks, with a resulting high number of context switches during execution. This is particularly problematic if the language does not support protected resources and hence servers must be used for all such entities. As will be illustrated in this chapter and Chapters 5 and 6, Ada, Java and C/Real-Time POSIX support the full range of entities.

4.3 Task representation

There are three basic mechanisms for representing concurrent execution: fork and join, cobegin and explicit task declaration.

4.3.1 Fork and join

This simple approach does not provide a visible entity for a task but merely supports two statements. The fork statement specifies that a designated routine should start executing concurrently with the invoker of the fork. The join statement allows the invoker to synchronize with the completion of the invoked routine. For example:

```
function F return ... is
begin
  ...
end F;

procedure P is
begin
  ...
  C:= fork F;
  .
  .
  .
  J:= join C;
  ...
end P;
```

Between the execution of the fork and the join, procedure P and function F will be executing concurrently. At the point of the join, the procedure will wait until the function has finished (if it has not already done so). Figure 4.3 illustrates the execution of fork and join.

The use of fork and join primitives is most prevalent in parallel programming languages (and they are often called spawn and join). The Linux operating system provides the clone system call that allows a new process to be created that shares the same address space as the parent process. The wait and waitpid system calls provide the join mechanism.

A version of fork and join can also be found in the C/Real-Time POSIX; here fork and vfork are used to create a copy of the invoker, and are used with the wait and waitpid system calls.

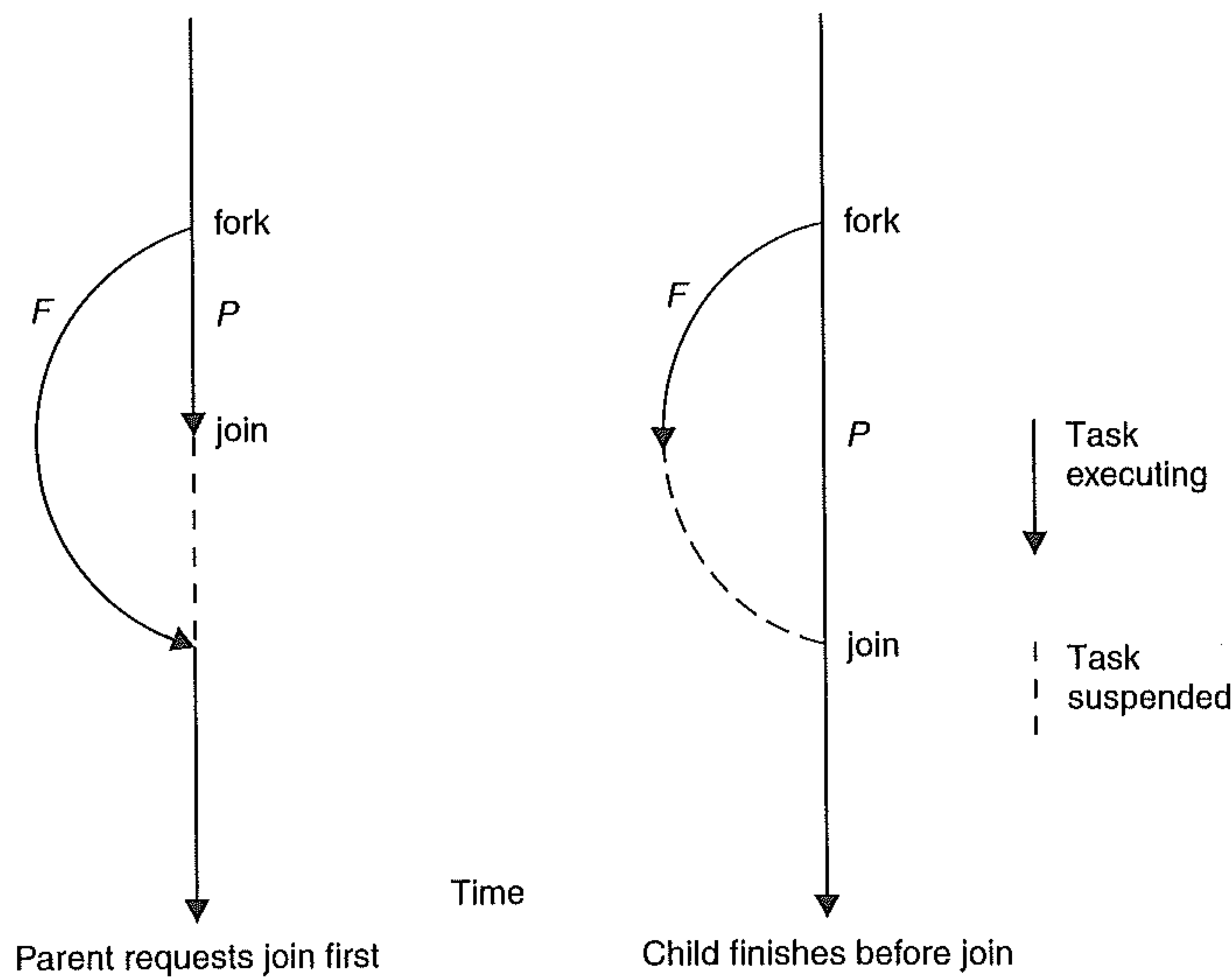


Figure 4.3 Fork and join.

Fork and join allow for dynamic process/task creation and provide a means of passing information to the child via parameters. Usually only a single value is returned by the child on its termination. Although flexible, fork and join do not provide a structured approach to process/task creation and can be error prone in use. For example, in some systems a guardian must explicitly ‘rejoin’ all dependants rather than merely wait for their completion.

4.3.2 Cobegin

The cobegin (or parbegin or par) is a structured way of denoting the concurrent execution of a collection of statements.

```
cobegin
  S1;
  S2;
  S3;
  .
  .
  .
  Sn
coend
```

This code causes the statements S1, S2 and so on to be executed concurrently. The cobegin statement terminates when all the concurrent statements have terminated. Each of the Si statements may be any construct allowed within the language, including simple assignments or procedure calls. If procedure calls are used, data can be passed to the invoked task via the parameters of the call. A cobegin statement could even include a sequence of statements that itself has a cobegin within it. In this way, a hierarchy of tasks can be supported. Figure 4.4 illustrates the execution of the cobegin statement.

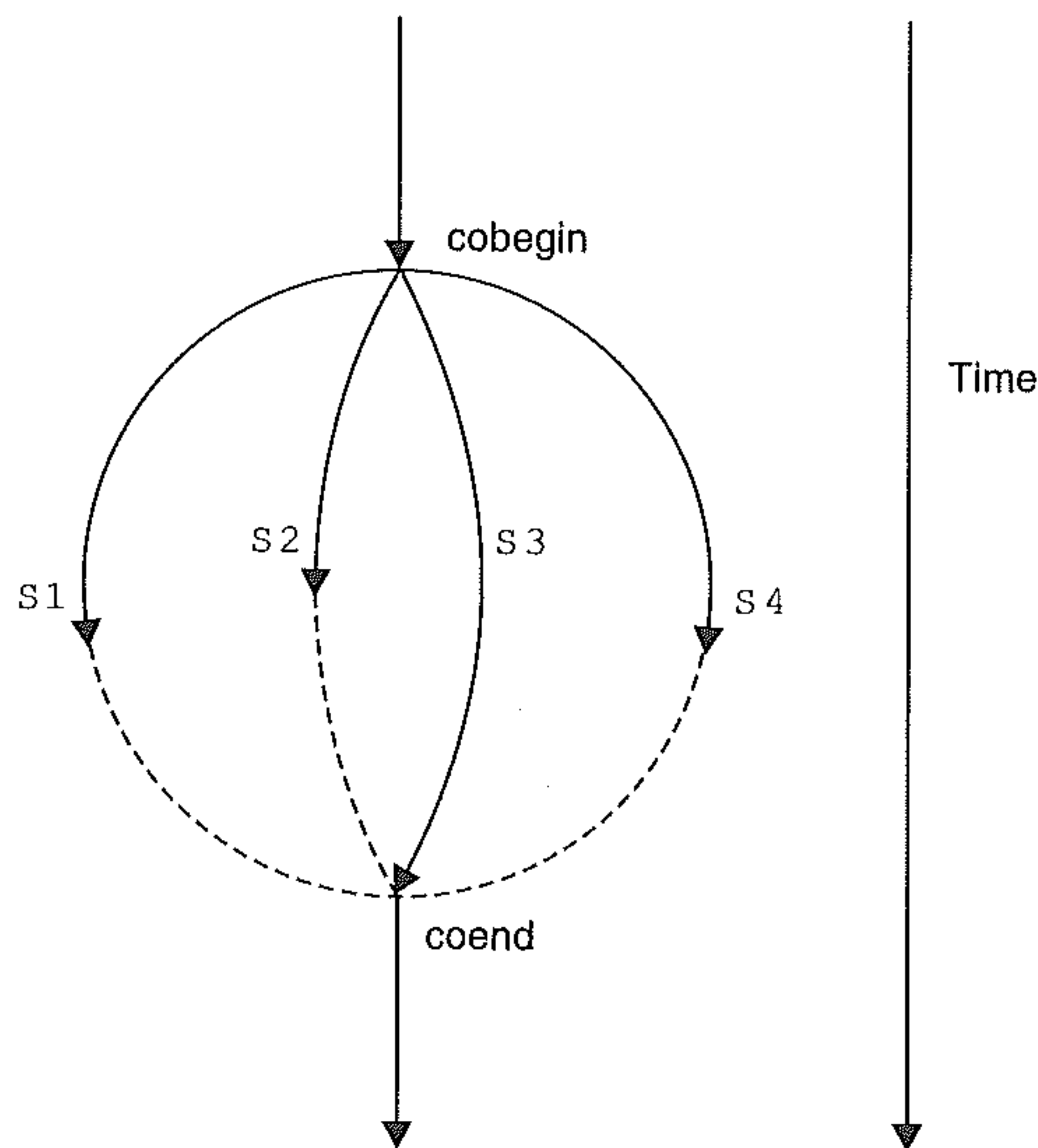


Figure 4.4 Cobegin.

Cobegin can be found in early concurrent programming languages (like Concurrent Pascal and occam2) but like 'fork and join' it has fallen from favour in modern real-time languages.

4.3.3 Explicit task declaration

Although sequential routines may be executed concurrently by means of the cobegin or fork, the structure of a concurrent program can be made much clearer if the routines themselves state whether they will be executed concurrently. Explicit task declaration provides such a facility, and has become the standard way of expressing concurrency in concurrent real-time languages. The following sections consider the mechanisms provided by Ada, Java and C/Real-Time POSIX.

4.4 Concurrent execution in Ada

The conventional unit of parallelism is called a **task** in Ada. Tasks may be declared at any program level; they are created implicitly upon entry to the scope of their declaration. The following example illustrates a procedure containing two tasks (A and B).

```

procedure Example1 is
  task A;
  task B;

  task body A is
    -- local declarations for task A
  begin
    -- sequence of statement for task A
  end A;

```

```

task body B is
  -- local declarations for task B
begin
  -- sequence of statements for task B
end B;

begin
  -- tasks A and B start their executions before
  -- the first statement of the sequence of
  -- statements belonging to the procedure
  .
  .
  .
end Example1;  -- the procedure does not terminate
                -- until tasks A and B have terminated.

```

Tasks consist of a specification and a body. They can be passed initialization data upon creation.

In the above, tasks A and B (which are created when the procedure is called) are said to have anonymous types, as they do not have types declared for them. Types could easily have been given for A and B:

```

task type A_Type;
task type B_Type;
A : A_Type;
B : B_Type;
task body A_Type is
  -- as before for task body A
task body B_Type is
  -- as before for task body B

```

With task types, a number of instances of the same task can easily be declared.

```

task type T;
A,B : T;

type Long is array (1..100) of T;

type Mixture is record
  Index : Integer;
  Action : T;
end record;

L : Long;
M : Mixture;

task body T is ...

```

To give a more concrete example of the use of tasks in an Ada program consider the implementation of a robot arm system. The arm can move in three dimensions. The movement in each dimension is powered by a separate motor that is controlled by a distinct Ada task. These tasks loop around, each reading a new relative setting for its dimension and then calling a low-level procedure `Move_Arm` to cause the arm to move.

First a package is declared that provides the necessary support types and sub-programs.

```
package Arm_Support is
  type Dimension is (Xplane, Yplane, Zplane);
  type Coordinate is new Integer range ...;

  procedure Move_Arm(D: Dimension; C: Coordinate);
    -- Moves the arm to C

  function New_Setting(D: Dimension) return Coordinate;
    -- Returns a new required relative position

end Arm_Support;
```

Now the main program can be presented.

```
with Arm_Support; use Arm_Support;
procedure Main is

  task type Control(Dim : Dimension);

  C1 : Control(Xplane);
  C2 : Control(Yplane);
  C3 : Control(Zplane);

  task body Control is
    Position : Coordinate;      -- current position
    Setting  : Coordinate;      -- required relative movement
  begin
    Position := Coordinate'First; -- rest position
  loop
    Move_Arm(Dim, Position);
    Setting := New_Setting(Dim);
    Position := Position + Setting;
  end loop;
end Control;
begin
  null;
end Main;
```

It is, perhaps, worth noting that Ada only allows discrete types and access (pointer) types to be passed as initialization parameters to a task.

By giving non-static values to the bounds of an array (of tasks), a dynamic number of tasks is created. Dynamic task creation can also be obtained explicitly using the 'new' operator on an access type (of a task type):

```
procedure Example2 is
  task type T;
  type A is access T;
  P : A;
  Q : A:= new T;
begin
  ...
  P := new T;
```

```

    Q := new T;
    ...
end Example2;

```

Q is declared to be of type A and is given a 'value' of a new allocation of T. This creates a task that immediately starts its initialization and execution; the task is designated Q.all (all is an Ada naming convention used to indicate the task itself, not the access pointer). During execution of the procedure, P is allocated a task (P.all) followed by a further allocation to Q. There are now three tasks active within the procedure; P.all, Q.all and the task that was created first. This first task is now anonymous as Q is no longer pointing to it. In addition to these three tasks, there is the task or main program executing the procedure code itself; in total, therefore, there are four distinct threads of control.

Tasks created by the operation of an allocator ('new') have the important property that the block that acts as its guardian (or **master** as Ada calls it) is not the block in which it is created but the one that contains the declaration of the access type. To illustrate this point, consider the following:

```

declare
    task type T;
    type A is access T;
begin
    .
    .
    .
    declare          -- inner block
        X : T;
        Y : A := new T;
    begin
        -- sequence of statements
    end; -- must wait for X to terminate but not Y.all
    .
    .      -- Y.all could still be active although the name Y is
    .      -- out of scope
    .
end; -- must wait for Y.all to terminate

```

Although both X and Y.all are created within the inner block, only X has this block as its master. Task Y.all is considered to be a dependent of the outer block, and it therefore does not affect the termination of the inner block.

If a task fails while it is being initialized (an exercise called **activation** in Ada) then the parent of that task has the exception `Tasking_Error` raised. This could occur, for example, if an inappropriate initial value is given to a variable. Once a task has started its true execution, it can catch any raised exceptions itself.

4.4.1 Task identification

One of the main uses of access variables is in providing another means of naming tasks. All task types in Ada are considered to be limited private. It is therefore not possible to pass a task by assignment to another data structure or program unit. For example, if

Robot_Arm and New_Arm are two variables of the same access type (the access type being obtained from a task type) then the following is illegal:

```
Robot_Arm.all := New_Arm.all;  -- not legal Ada
```

However,

```
Robot_Arm := New_Arm;
```

is quite legal and means that Robot_Arm is now designating the same task as New_Arm. Care must be exercised here, as duplicated names can cause confusion and lead to programs that are difficult to understand. Furthermore, tasks may be left without any access pointers; such tasks are said to be **anonymous**. For example, if Robot_Arm pointed to a task, when it was overwritten with New_Arm, the previous task will have become anonymous if there were no other pointers to it.

In some circumstances it is useful for a task to have a unique identifier (rather than a name). For example, a server task is not usually concerned with the type of the client tasks. Indeed, when communication and synchronization are discussed in the next chapter, it will be seen that the server has no direct knowledge of who its clients are. However, there are occasions when a server needs to know that the client task it is communicating with is the same client task that it previously communicated with. Although the core Ada language provides no such facility, the Systems Programming Annex provides a mechanism by which a task can obtain its own unique identification. This can then be passed to other tasks. An abridged version of the associated package is shown in Program 4.1.

As well as this package, the Annex supports two attributes.

- (1) For any prefix T of a task type, T'Identity returns a value of type Task_Id that equals the unique identifier of the task denoted by T.
- (2) For any prefix E that denotes an entry declaration, E'Caller returns a value of type Task_Id that equals the unique identifier of the task whose entry call is being serviced. The attribute is only allowed inside an entry body or an accept statement (see Chapters 5 and 6).

Program 4.1 The Ada.Task_Identification package.

```
package Ada.Task_Identification is
  type Task_Id is private;
  Null_Task_Id : constant Task_Id;

  function "=" (Left, Right : Task_Id) return Boolean;

  function Current_Task return Task_Id;
    -- returns unique id of calling task

  -- other functions not relevant to this discussion
private
  ...
end Ada.Task_Identification;
```

Care must be taken when using task identifiers since there is no guarantee that, at some later time, the task will still be active or even in scope.

4.4.2 Task termination

Having considered creation and representation, one is left with task termination. Ada provides a range of options; a task will terminate if:

- (1) it completes execution of its body (either normally or as the result of an unhandled exception);
- (2) it executes a 'terminate' alternative of a select statement (this is explained in Section 6.5) thereby implying that it is no longer required;
- (3) it is aborted.

If an unhandled exception has caused the task's demise then the effect of the error is isolated to just that task.

Another task can enquire (by the use of an attribute) if a task has terminated:

```
if T'Terminated then    -- for some task T
    -- error recovery action
end if;
```

However using this mechanism, the enquiring task cannot differentiate between normal or error termination of the other task, and, of course, the task could terminate just after the test has been performed.

In Ada 2005, extra support has been added to help the program manage task termination, in particular unexpected termination due to error conditions. An abridged version of the package is shown in Program 4.2. Essentially, a task can have an associated access variable to a protected procedure.¹ The Ada run-time support system will call this procedure when the task terminates. A parameter to the call gives the cause of the termination.

4.4.3 Task abortion

Any task can abort any other task whose name is in scope. When a task is aborted, all its dependants are also aborted. The abort facility allows wayward tasks to be removed. It is, of course, a very dangerous mechanism and should only be used when there is no alternative course of action (see Section 7.6.1).

4.4.4 OOP and concurrency

Ada 95 did not attempt to integrate the language's support for concurrent programming directly into the OOP model. Instead, the models were orthogonal and paradigms had to be created to allow the benefits of OOP to be available in a concurrent environment. Since Ada 95, object-oriented programming techniques have advanced and the notion of an *interface* has emerged as the preferred mechanism for acquiring many of the benefits of multiple inheritance without most of the problems. The introduction of this mechanism

¹ A protected procedure in Ada is a procedure that is declared within a monitor-like object, see Section 5.8.

Program 4.2 The Ada 2005 support for task termination.

```

with Ada.Task_Identification; with Ada.Exceptions;

package Ada.Task_Termination is
  ...
  type Cause_Of_Termination is (Normal, Abnormal,
                                Unhandled_Exception);

  type Termination_Handler is access protected procedure
    (Cause : in Cause_Of_Termination;
     T      : in Ada.Task_Identification.Task_Id;
     X      : in Ada.Exceptions.Exception_Occurrence);

  procedure Set_Specific_Handler
    (T      : in Ada.Task_Identification.Task_Id;
     Handler : in Termination_Handler);

  function Specific_Handler (T : Ada.Task_Identification.Task_Id)
    return Termination_Handler;
end Ada.Task_Termination;

```

into Ada 2005 allows tasks to support interfaces. Whilst this does not give the full power of extensible tasks, it does give much of the functionality.

The interface facility provided by Ada 2005 is related to inter-task communication, and will be considered in detail in Sections 5.8.2 and 6.3.3.

4.5 Concurrent execution in Java

Java has a predefined class, `java.lang.Thread` which provides the mechanism by which threads (tasks) are created. However, to avoid all threads having to be child classes of `Thread`, Java also has a standard interface, called `Runnable`.

```

public interface Runnable {
  public void run();
}

```

Hence any class which wishes to express concurrent execution must implement this interface and provide the `run` method. The `Thread` class given in Program 4.3 does just this.

`Thread` is a subclass of `Object`. It provides several constructor methods and a `run` method. Using these constructors, threads can be created in two ways.

The first is to declare a class to be a subclass of `Thread` and override the `run` method. An instance of the subclass can then be allocated and started. For instance, in the robot arm example assume that the following classes and objects are available:

```

public class UserInterface {
  public int newSetting (int Dim) { ... }
  ...
}

```

Program 4.3 An abridged version of the Java Thread class.

```

package java.lang;
public class Thread extends Object implements Runnable {

    //nested classes
    public static final enum State {BLOCKED, NEW, RUNNABLE,
                                    TERMINATED, TIMED_WAITING, WAITING};

    public static interface UncaughtExceptionHandler{
        public void uncaughtException(Thread t, Throwable e);
    };

    // constructors
    public Thread();
    public Thread(String name);
    public Thread(Runnable target);
    public Thread(Runnable target, String name);

    // methods associated with thread execution
    public void run();
    public void start();
    public final void join() throws InterruptedException;

    // methods associated with thread naming and identification
    public static Thread currentThread();
    public String getName();
    public void setName(String name);

    // methods associated with thread states and termination
    public static UncaughtExceptionHandler
        getDefaultUncaughtExceptionHandler();
    public UncaughtExceptionHandler
        getUncaughtExceptionHandler();
    public static setDefaultUncaughtExceptionHandler1(
        UncaughtExceptionHandler eh);
    public void setUncaughtExceptionHandler(
        UncaughtExceptionHandler);
    public final boolean isAlive();
    public final boolean isDaemon();
    public final void setDaemon();
    public State getState();

    // other methods etc will be introduced throughout this
    // book

    // Note, RuntimeExceptions are not listed as part of the
    // method specification.
}

```

```
public class Arm {
    public void move(int dim, int pos) { ... }
}
```

```
UserInterface UI = new UserInterface(); Arm Robot = new Arm();
```

Given the above classes, in scope, the following will declare a class which can be used to represent the three controllers.

```
public class Control extends Thread {

    private int dim;

    public Control(int Dimension) { // constructor
        super();
        dim = Dimension;
    }

    public void run() {

        int position = 0;
        int setting;

        while(true)
        {
            Robot.move(dim, position);
            setting = UI.newSetting(dim);
            position = position + setting;
        }
    }
}
```

The three controllers can now be created:

```
final int xPlane = 0; // final indicates a constant
final int yPlane = 1;
final int zPlane = 2;

Control C1 = new Control(xPlane);
Control C2 = new Control(yPlane);
Control C3 = new Control(zPlane);
```

At this point, the threads have been created, any variables declared have been initialized and the constructor methods for the `Control` and `Thread` class have been called (Java calls this the *new* state). However, the thread does not begin its execution until the `start` method is called:

```
C1.start();
C2.start();
C3.start();
```

Note that if the `run` method is called explicitly then the code is executed sequentially.

The second way to create a thread is to declare a class that implements the `Runnable` interface. An instance of the class can then be allocated and passed as

an argument during the creation of a thread object. Remember, Java threads are not created automatically when their associated objects are created, but must be explicitly created and started using the `start` method.

```
public class Control implements Runnable {

    private int dim;

    public Control(int Dimension) { // constructor
        dim = Dimension;
    }

    public void run() {
        int position = 0;
        int setting;

        while(true) {
            Robot.move(dim, position);
            setting = UI.newSetting(dim);
            position = position + setting;
        }
    }
}
```

The three controllers can now be created:

```
final int xPlane = 0;
final int yPlane = 1;
final int zPlane = 2;

Control C1 = new Control(xPlane); // no thread created yet
Control C2 = new Control(yPlane);
Control C3 = new Control(zPlane);
```

and then associated with threads and started:

```
// constructors passed a Runnable interface and threads created
Thread X = new Thread(C1);
Thread Y = new Thread(C2);
Thread Z = new Thread(C2);

X.start(); // thread started
Y.start();
Z.start();
```

Like Ada, Java allows dynamic thread creation. In contrast to Ada, Java (by means of constructor methods) allows arbitrary data to be passed as parameters (of course, all objects are passed by references which are similar to Ada's access variables).

Although Java allows thread hierarchies and thread groups to be created, there is no master or guardian concept. This is because Java relies on garbage collection to clean up objects which can no longer be accessed. The exception to this is the main program. The main program in Java terminates when all its user threads have terminated.

One thread can wait for another thread (the target) to terminate by issuing the `join` method call on the target's thread object. Furthermore, the `isAlive` method allows a thread to determine if the target thread has terminated.

4.5.1 Thread identification

There are two ways that threads can be identified. If the code of the thread is a subclass of the `Thread` class, then the following will define a thread identifier.

```
Thread threadID;
```

Any subclass of `Thread` can be assigned to this object (because of the reference semantics of Java). Note that where the code of a thread is passed through a constructor with the `Runnable` interface, the thread identifier would be the thread object not the object providing the `Runnable` interface. To define an identifier which would identify the object providing the code requires a `Runnable` interface to be defined.

```
Runnable threadCodeID;
```

However, once a reference to the `Runnable` object has been obtained there is little explicitly that can be done with it. All the thread-related operations are provided by the `Thread` class.

The identity of the currently running thread can be found using the `currentThread` method. This method has a `static` modifier which means that there is only one method for all instances of `Thread` objects. Hence the method can always be called using the `Thread` class.

4.5.2 Thread termination

A Java thread terminates when it completes execution of its `run` method either normally or as the result of an unhandled exception.

Threads can be of two types: **user** threads or **daemon** threads. Daemon threads are those threads which provide general services and typically never terminate. Hence when all user threads have terminated, daemon threads can also be terminated and the main program terminates. (Daemon threads provide the same functionality as the Ada 'or terminate' option on the `select` statement – see Section 6.5.) The `setDaemon` method is used to identify daemon threads, but must be called before the thread is started.

Early versions of the language did provide the equivalent of the Ada abort facility. However, these mechanisms have been deprecated as Java views them as being inherently unsafe.

4.5.3 Thread-related exceptions

In Chapter 3, Java `RuntimeExceptions` were introduced. The following `RuntimeExceptions` are relevant to threads.

The `IllegalThreadStateException` is thrown when:

- the `start` method is called and the thread has already been started;
- the `setDaemon` method has been called and the thread has already been started.

The `InterruptedException` is thrown if a thread which has issued a `join` method is woken up by the thread being interrupted rather than the target thread terminating (see Section 7.7.2).

As of Java 5, all threads can set uncaught exception handlers. Essentially, this is a handler that is run if the thread terminates due to an uncaught exception. (It is equivalent to the Ada `Task_Termination` package given in Program 4.2.) The definition of the handler is via the static `UncaughtExceptionHandler` interface included locally in the `Thread` class. It contains a single method `uncaughtException`. An object implementing this interface can be passed to the static method `setDefaultUncaughtExceptionHandler`. The Java Virtual Machine will call the method in this object before the associated thread terminates.

4.5.4 Real-time threads

The Java programming language lacks many facilities for programming real-time systems; hence the development of Real-Time Java. Real-Time Java produces several new thread classes. These will be considered in Section 10.1.

4.6 Concurrent execution in C/Real-Time POSIX

C/Real-Time POSIX provides three mechanisms for creating concurrent activities. The first is the traditional process-level Unix `fork` mechanism (and its associated `wait` system call). This causes a copy of the entire process to be created and executed. The details of `fork` can be found in most textbooks on operating systems and will not be discussed here. (An example of process creation using `fork` is given in Section 6.7.) The second is the `spawn` system call which is equivalent in function to a combined `fork` and `exec`.

C/Real-Time POSIX also allows for each process to contain several ‘threads’ of execution. These threads all have access to the same memory locations and run in a single address space. Thus, they can be compared with Ada’s tasks and Java’s threads. Programs 4.4 and 4.5 illustrate the primary C interface for thread creation in Real-Time POSIX.

It is not defined whether a POSIX-compliant system must support a single kernel-level thread (see Section 4.1) for each application-level thread created by a call to the `pthread_create` API function. Usually, a kernel will ensure that a sufficient number of threads can execute in order to ensure an application can make progress. The `pthread_setconcurrency` function allows an application to inform the kernel of its desired level of concurrency (i.e. how many kernel threads it would like – a zero means that the kernel should manage the level itself). However, there is no requirement for the kernel to take notice of this request.

All threads have attributes (for example, their stack size and any overflow buffer – called a **guard** in C/Real-Time POSIX). To manipulate these attributes, it is necessary to define an attribute object (of type `pthread_attr_t`) and then call functions to set and get the attributes. Once the correct attribute object has been established, a thread can be created and the appropriate attribute object passed. Program 4.4 shows the typical interfaces.

Program 4.4 A C/Real-Time POSIX interface to thread attributes.

```

typedef ... pthread_t;  /* details not defined */
typedef ... pthread_attr_t;
typedef ... size_t;

int pthread_attr_init(pthread_attr_t *attr);
    /* initializes a thread attribute pointed at by attr to
       their default values */

int pthread_attr_destroy(pthread_attr_t *attr);
    /* destroys a thread attribute pointed at by attr*/

int pthread_attr_setstacksize(pthread_attr_t *attr,
                               size_t stacksize);
    /* set the stack size of a thread attribute */

int pthread_attr_getstacksize(const pthread_attr_t *attr,
                               size_t *stacksize);
    /* get the stack size of a thread attribute */

int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);
    /* set the detach state of the attribute */

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
    /* get the detach state of the attribute */

int pthread_attr_setguardsize(pthread_attr_t *attr,
                               size_t guardsize);
    /* set the guard size of a thread attribute */

int pthread_attr_getguardsize(const pthread_attr_t *attr,
                               size_t *guardsize);
    /* get the guard size of a thread attribute */

...
/* other attributes associated with scheduling */

/* Unless otherwise stated, all the above integer functions
   returns 0 if successful, otherwise an error number is returned
*/

```

Every created thread has an associated identifier (of type `pthread_t`) that is unique with respect to other threads in the same process. A thread can obtain its own identifier (via `pthread_self`).

A thread becomes eligible for execution as soon as it is created by `pthread_create`; there is no equivalent to the Ada activation state or the Java new state. This function takes four pointer arguments: a thread identifier (returned by the call), a set of attributes, a function that represents the code of the thread when it executes, and the set of parameters to be passed to this function when it is called. The thread can terminate

Program 4.5 A C Real-Time POSIX interface to threads.

```

typedef ... pthread_t;  /* details not defined */
typedef ... pthread_attr_t;

int pthread_getconcurrency();
    /* returns the last set value of pthread_getconcurrency */

int pthread_setconcurrency(int level);
    /* sets the application's preferred thread concurrency level;
       returns the old level */

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start_routine)(void *), void *arg);
    /* create a new thread with the given attributes and call the
       given start_routine with the given argument */

int pthread_join(pthread_t thread, void **value_ptr);
    /* suspends the calling thread until the named thread has
       terminated, any returned values are pointed at by value_ptr */

void pthread_exit(void *value_ptr);
    /* terminate the calling thread and make the pointer value_ptr
       available to any joining thread */

int pthread_detach(pthread_t thread);
    /* the storage space associated with the given thread may be
       reclaimed when the thread terminates */

pthread_t pthread_self(void);
    /* return the thread id of the calling thread */

int pthread_equal(pthread_t t1, pthread_t t2);
    /* compare two thread ids
       return non 0 if equal, 0 otherwise */

int pthread_atfork(void (*prepare)(void), void (*parent)(void),
                  void (*child)(void));
    /* used for managing the resources shared by a multi-threaded
       program when a fork is performed */

/* Unless otherwise stated, all the above integer functions
   returns 0 if successful, otherwise an error number is returned
*/

```

normally by returning from its `start_routine` or by calling `pthread_exit` or by receiving a signal sent to it (see Section 7.5.1). It can also be aborted by the use of `pthread_cancel`. One thread can wait for another thread to terminate via the `pthread_join` function.

Finally, the activity of cleaning up after a thread's execution and reclaiming its storage is termed **detaching**. There are two ways to achieve this: by calling the `pthread_join` function and waiting until the thread terminates, or by setting the

detached attribute of the thread (either at creation time or dynamically by calling the `pthread_detach` function). If the detached attribute is set, the thread is not joinable and its storage space may be reclaimed automatically when the thread terminates.

In many ways, the interface provided by C/Real-Time POSIX threads is similar to that which would be used by a compiler to interface to its run-time support systems. Indeed, the run-time support system for Ada may well be implemented using POSIX threads. The advantage of providing higher-level language abstractions (such as those of Ada and Java) is that it removes the possibility of errors in using the interface.

To illustrate the simple use of the C/Real-Time POSIX thread creation facility, the robot arm program is shown below.

```
#include <pthread.h>

pthread_attr_t attributes;
pthread_t xp, yp, zp;

typedef enum {xplane, yplane, zplane} dimension;

int new_setting(dimension D);
void move_arm(dimension D, int P);

void controller(dimension *dim) {
    int position, setting;

    position = 0;
    while (1) {
        move_arm(*dim, position);
        setting = new_setting(*dim);
        position = position + setting;
    }
    /* note, no call to pthread_exit, process does not terminate */
}

#include <stdlib.h>
int main() {
    dimension X, Y, Z;
    void *result;

    X = xplane,
    Y = yplane;
    Z = zplane;
    if(pthread_attr_init(&attributes) != 0)
        /* set default attributes */
        exit(EXIT_FAILURE);
    if(pthread_create(&xp, &attributes,
                    (void *)controller, &X) != 0)
        exit(EXIT_FAILURE);
    if(pthread_create(&yp, &attributes,
                    (void *)controller, &Y) != 0)
        exit(EXIT_FAILURE);
    if(pthread_create(&zp, &attributes,
                    (void *)controller, &Z) != 0)
        exit(EXIT_FAILURE);
    pthread_join(xp, (void **)&result);
```

```

    /* need to block main program */

    exit(EXIT_FAILURE);
    /* error exit, the program should not terminate */
}

```

A thread attribute object is created with the default attributes. Calls to `pthread_create` create each instance of the controller thread and pass a parameter indicating its domain of operation. If any errors are returned from the operating system, the program terminates by calling the `exit` routine. Note that a program which terminates with threads still executing results in those threads being terminated. Hence it is necessary for the main program to issue a `pthread_join` system call even though the threads do not terminate.

In Section 3.1.1, a style of handling the error returns from POSIX was given. It assumes that each call has a macro defined which tests the return value and calls, if appropriate, an error-handling routine. It is therefore possible to write the above code in the following more readable way:

```

int main() {
    dimension X, Y, Z;
    void *result;

    X = xplane,
    Y = yplane;
    Z = zplane;

    PTHREAD_ATTR_INIT(&attributes);

    PTHREAD_CREATE(&xp, &attributes, (void *)controller, &X);
    PTHREAD_CREATE(&yp, &attributes, (void *)controller, &Y);
    PTHREAD_CREATE(&zp, &attributes, (void *)controller, &Z);

    PTHREAD_JOIN(xp, (void **)&result);
}

```

Finally, it should be noted that conceptually forking a process (program) which contains multiple threads is not straightforward, as some threads in the process may hold resources or be executing system calls. The C/Real-Time POSIX standard specifies that the child process will only have one thread (see POSIX 1003 (Open Group/IEEE, 2004)). The `pthread_atfork` function allows a process to specify three functions that can be called to help the programmer manage such situations.

- `prepare` – this routine is called immediately *before* the fork is called.
- `parent` – this routine is called in the parent immediately after the fork has returned.
- `child` – this routine is called in the child immediately after the fork has returned.

Typically, the `prepare` will try to obtain all the shared locks (therefore, blocking the calling threads until the other threads have released the shared locks). The `parent` and `child` routine will typically release the locks.

4.7 Multiprocessor and distributed systems

Most concurrent programming languages usually attempt to define their semantics so that they can be implemented on single processor or multiprocessor system where there is access to global shared memory. In the multiple processor case, it is usual to assume that all processors in the system can execute all tasks. Hence, when a task is dispatched it can be dispatched to any of the available processors. This is sometimes called **global dispatching**. Hence during its lifetime, a task may migrate from processor to processor.

From a real-time perspective, predictability is a major concern. Fixing tasks to processors and not allowing them to migrate usually results in more predictable response times (see Section 11.14). Hence, some operating systems (for example Linux) provide an API that allows threads to be constrained to execute on a limited set of processors. This is usually called **processor affinity**.

Whilst tasks can be mapped to different processors in a distributed system, it is more usual to provide some other form of encapsulation method to represent the 'unit' of distribution. For example, processes, objects, partitions, agents and guardians have all been proposed as units of distribution. All these constructs provide well-defined interfaces which allow them to encapsulate local resources and provide remote access.

Hence, the production of an application to execute on a distributed system involves several steps which are not required when programs are produced for a single or multiprocessor platform.

- **Partitioning** is the activity of dividing the system into parts (units of distribution) suitable for placement onto the processing nodes of the target system.
- **Configuration** takes place when the partitioned parts of the program are associated with particular processing elements in the target system.
- **Allocation** covers the actual activity of turning the configured system into a collection of executable modules and downloading these to the processing elements of the target system.
- **Transparent execution** is the execution of the distributed software so that remote resources can be accessed in a manner which is independent of their location – usually using some form of message passing.
- **Reconfiguration** is the dynamic change to the location of a software component or resource.

Languages which have been designed explicitly to address distributed programming will provide linguistic support for at least the partitioning stage of system development. Some approaches will allow configuration information to be included in the program source, whereas others will provide a separate **configuration** language. Allocation and reconfiguration, typically, require support from the programming support environment and operating system.

The support that Ada, Java and C/Real-Time POSIX provide for multiprocessor and distributed systems is considered in the following subsections.

4.7.1 Ada

The Ada Reference Manual allows a program's implementation to be on a multiprocessor system. However, it provides no direct support that allows programmers to partition their tasks onto the processors in the given system. The following ARM quote illustrates this.

NOTES 1 Concurrent task execution may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor. On the other hand, whenever an implementation can determine that the required semantic effects can be achieved when parts of the execution of a given task are performed by different physical processors acting in parallel, it may choose to perform them in this way. *ARM Section 9 par 11.*

This simply allows multiprocessor execution and also allows parallel execution of a single task if it can be achieved, in effect, 'as if executed sequentially'.

In a multiprocessor system, a task can be on the ready queues of more than one processor. At the extreme, if several processors share the same set of ready tasks, the contents of their ready queues is identical, and so they can be viewed as sharing one ready queue, and can be implemented that way. Thus, the dispatching model covers multiprocessors where dispatching is implemented using a single ready queue, as well as those with separate dispatching domains. *ARM Section 10.1/2 par 15.*

This allows the full range of dispatching identified above. However, currently the only way that an implementation can provide the mechanisms to allow the programmer to set the processor affinity of their tasks is via implementation-defined pragmas, or non-standard library packages.

Ada defines a distributed system as an

interconnection of one or more processing nodes (a system resource that has both computational and storage capabilities), and zero or more storage nodes (a system resource that has only storage capabilities, with the storage addressable by more than one processing nodes).

The Ada model for programming distributed systems specifies a **partition** as the unit of distribution. Partitions comprise aggregations of library units (separately compiled library packages or subprograms) that collectively may execute in a distributed target execution environment. The configuration of library units into partitions is not defined by the language; it is assumed that an implementation will provide this, along with facilities for allocation and, if necessary, reconfiguration.

Each partition resides at a single execution site where all its library units occupy the same logical address space. More than one partition may, however, reside on the same execution site.

Partitions may be either **active** or **passive**. The library units comprising an active partition reside and execute upon the same processing element. In contrast, library units comprising a passive partition reside at a storage element that is directly accessible to the nodes of different active partitions that reference them. This model ensures that active

partitions cannot directly access variables in other active partitions. Variables can only be shared directly between active partitions by encapsulating them in a passive partition. Communication between active partitions is defined in the language to be via remote subprogram calls (however, an implementation may provide other communication mechanisms).

Categorization pragmas

To aid the construction of distributed programs, Ada distinguishes between different categories of library units, and imposes restrictions on these categories to maintain type consistency across the distributed program. The categories (some of these are useful in their own right, irrespective of whether the program is to be distributed) are designated by the following pragmas.

- **Preelaborate** – a preelaboratable library unit is one that can be elaborated without execution of code at run-time.
- **Pure** – pure packages are preelaboratable packages with further restrictions which enable them to be freely replicated in different active or passive partitions without introducing any type inconsistencies. These restrictions concern the declaration of objects and types; in particular, variables and named access types are not allowed unless they are within a subprogram, task unit or protected unit.
- **Remote.Types** – a `Remote.Types` package is a preelaboratable package that must not contain any variable declarations within the visible part.
- **Shared.Passive** – `Shared.Passive` library units are used for managing global data shared between active partitions. They are, therefore, configured on storage nodes in the distributed system.

4.7.2 Java

As with Ada, Java's semantics allow for the parallel execution of threads on a shared memory multiprocessor system and does not support processor affinity. However, the language at least allows the determination of the number of processors on which the application is executing (via the `availableProcessors` method in the `Runtime` class). The full meaning of the execution of Java threads on a shared memory multiprocessor system is defined via the Java Memory Model. This will be considered in Section 5.10.1.

There are essentially two ways in which to construct distributed Java applications.

- (1) Execute Java programs on separate machines and use the Java networking facilities.
- (2) Use remote objects.

Java networking

The two prominent network communication protocols in use today are UDP and TCP. The Java environment provides classes (in the `java.net` package) which allow easy access to them. The API to these protocols is via the `Socket` class (for the reliable TCP

protocol) and the `DatagramSocket` class (for the UDP protocol). It is beyond the scope of this book to consider this approach in detail – see the Further Reading section at the end of this chapter for alternative sources of information.

Remote objects

Although Java provides a convenient way of accessing network protocols, these protocols are still complex and are a deterrent to writing distributed applications. Consequently, Java supports a distributed object communication model through the notion of **remote objects**. This will be discussed further in Section 6.8.4.

4.7.3 C/Real-Time POSIX

C/Real-Time POSIX defines the ‘Scheduling Allocation Domain’ as the set of processors on which an individual thread can be scheduled at any given time. C/Real-Time POSIX states that (Open Group/IEEE, 2004):

- ‘For application threads with scheduling allocation domains of size equal to one, the scheduling rules defined for `SCHED_FIFO` and `SCHED_RR` shall be used.’
- ‘For application threads with scheduling allocation domains of size greater than one, the rules defined for `SCHED_FIFO`, `SCHED_RR`, and `SCHED_SPORADIC` shall be used in an implementation-defined manner.’
- ‘The choice of scheduling allocation domain size and the level of application control over scheduling allocation domains is implementation-defined. Conforming implementations may change the size of scheduling allocation domains and the binding of threads to scheduling allocation domains at any time.’

The details of the actual scheduling algorithms are of no concern here (they will be covered in Chapter 12). With this approach, there is no portable way to specify an affinity between threads and processors.

4.7.4 Processor affinity

From the above, it is clear that neither Ada, Java or C/Real-Time POSIX allows programmer control over the mapping of tasks to processors in a shared memory multiprocessor system. To give a flavour of what could be provided in future by these languages, consider the support that Linux provides.

Since Kernel version 2.5.8, Linux has provided support for multiprocessor systems (Linux Manual Page, 2006) via the notion of CPU affinity. Each process in the system can have its CPU affinity set according to a CPU affinity mask. A process’s CPU affinity mask determines the set of CPUs on which it is eligible to run.

```
#include <sched.h>

int sched_setaffinity(pid_t pid,
    unsigned int cpusetsize, cpu_set_t *mask);

int sched_getaffinity(pid_t pid,
    unsigned int cpusetsize, cpu_set_t *mask);
```



```

void CPU_CLR(int cpu, cpu_set_t *set);

int CPU_ISSET(int cpu, cpu_set_t *set);

void CPU_SET(int cpu, cpu_set_t *set);

void CPU_ZERO(cpu_set_t *set);

```

A CPU affinity mask is represented by the `cpu_set_t` structure, a 'CPU set', pointed to by the mask. Four macros are provided to manipulate CPU sets. `CPU_ZERO` clears a set. `CPU_SET` and `CPU_CLR` respectively add and remove a given CPU from a set. `CPU_ISSET` tests to see if a CPU is part of the set. The first available CPU on the system corresponds to a CPU value of 0, the next CPU corresponds to a CPU value of 1, and so on. A constant `CPU_SETSIZE` (1024) specifies a value one greater than the maximum CPU number that can be stored in a CPU set.

The `sched_setaffinity` method sets the CPU affinity mask of the process, whose ID is `pid`, to the value specified by `mask`. If the process specified by `pid` is not currently running on one of the CPUs specified in `mask`, then that process is migrated to one of the CPUs specified in `mask`.

The `sched_getaffinity` method allows the current mask to be obtained.

The affinity mask is actually a per-thread attribute that can be adjusted independently for each of the threads in a thread group.

4.8 A simple embedded system

In order to illustrate some of the advantages and disadvantages of concurrent programming, a simple embedded system will now be considered. Figure 4.5 outlines this simple system: a task *T* takes readings from a set of thermocouples (via an analog-to-digital converter, ADC) and makes appropriate changes to a heater (via a digitally controlled switch). Process *P* has a similar function, but for pressure (it uses a digital-to-analog converter, DAC). Both *T* and *P* must communicate data to *S*, which presents measurements to an operator via a console. Note that *P* and *T* are active; *S* is a resource (it just responds to requests from *T* and *P*): it may be implemented as a protected resource or a server if it interacts more extensively with the user.

The overall objective of this embedded system is to keep the temperature and pressure of some chemical process within defined limits. A real system of this type would clearly be more complex – allowing, for example, the operator to change the limits. However, even for this simple system, the implementation could take one of three forms.

- (1) A single program is used which ignores the logical concurrency of *T*, *P* and *S*. No operating system support is required.
- (2) *T*, *P* and *S* are written in a sequential programming language (either as separate programs or distinct procedures in the same program) and operating system primitives are used for program/process creation and interaction.
- (3) A single concurrent program is used which retains the logical structure of *T*, *P* and *S*. No direct operating system support is required by the program although a run-time support system is needed.

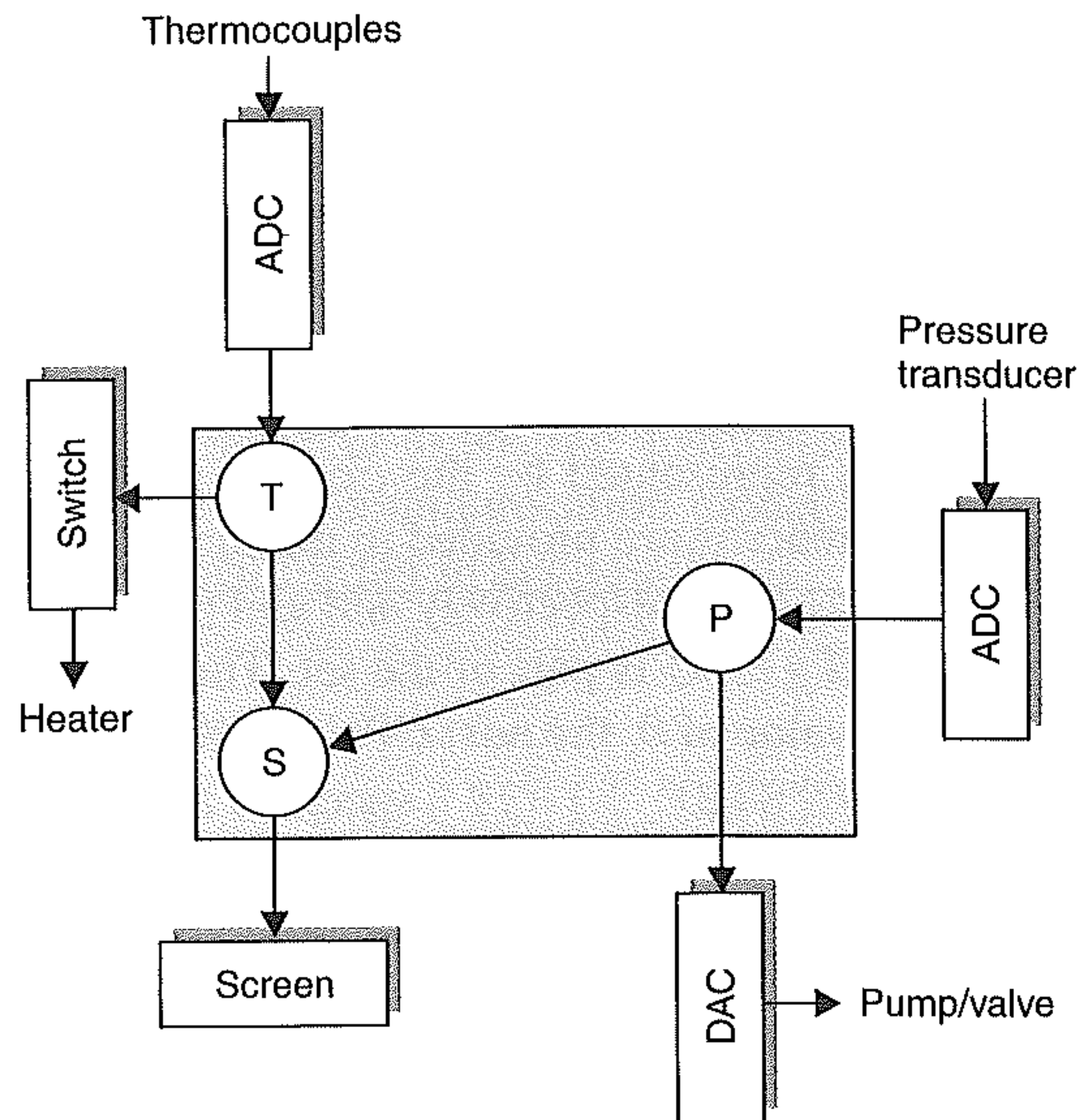


Figure 4.5 A simple embedded system.

To illustrate these solutions, consider the Ada code to implement the simple embedded system. In order to simplify the structure of the control software, the following passive packages will be assumed to have been implemented:

```

package Data_Types is
  -- necessary type definitions
  type Temp_Reading is new Integer range 10..500;
  type Pressure_Reading is new Integer range 0..750;
  type Heater_Setting is (On, Off);
  type Pressure_Setting is new Integer range 0..9;
end Data_Types;

with Data_Types; use Data_Types;
package IO is
  -- procedures for data exchange with the environment
  procedure Read(TR : out Temp_Reading); -- from ADC
  procedure Read(PR : out Pressure_Reading);
    -- note, this is an example of overloading; two reads
    -- are defined but they have a different parameter type;
    -- this is also the case with the following writes
  procedure Write(HS : Heater_Setting); -- to switch.
  procedure Write(PS : Pressure_Setting); -- to DAC
  procedure Write(TR : Temp_Reading); -- to console
  procedure Write(PR : Pressure_Reading); -- to console
end IO;

with Data_Types; use Data_Types;
package Control_Procedures is
  -- procedures for converting a reading into
  -- an appropriate setting for output to
  
```



```

procedure Temp_Convert (TR : Temp_Reading;
                        HS : out Heater_Setting);
procedure Pressure_Convert (PR : Pressure_Reading;
                           PS : out Pressure_Setting);
end Control_Procedures;

```

Sequential solution

A simple sequential control program could have the following structure:

```

with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Controller is
  TR : Temp_Reading;
  PR : Pressure_Reading;
  HS : Heater_Setting;
  PS : Pressure_Setting;
begin
  loop
    Read(TR);      -- from ADC
    Temp_Convert(TR,HS); -- convert reading to setting
    Write(HS);     -- to switch
    Write(TR);     -- to console
    Read(PR);      -- as above for pressure
    Pressure_Convert (PR,PS);
    Write(PS);
    Write(PR);
  end loop; -- infinite loop, common in embedded software
end Controller;

```

This code has the immediate handicap that temperature and pressure readings must be taken at the same rate, which may not be in accordance with requirements. The use of counters and appropriate **if** statements will improve the situation, but it may still be necessary to split the computationally intensive sections (the conversion procedures `Temp_Convert` and `Pressure_Convert`) into a number of distinct actions, and interleave these actions so as to meet a required balance of work. Even if this were done, there remains a serious drawback with this program structure: while waiting to read a temperature no attention can be given to pressure (and vice versa). Moreover, if there is a system failure that results in, say, control never returning from the temperature `Read`, then in addition to this problem, no further pressure `Reads` would be taken.

An improvement on this sequential program can be made by including two boolean functions in the package `IO`, `Ready_Temp` and `Ready_Pres`, to indicate the availability of an item to read. The control program then becomes:

```

with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Controller is
  TR : Temp_Reading;
  PR : Pressure_Reading;
  HS : Heater_Setting;

```



```

    PS : Pressure_Setting;
    Ready_Temp, Ready_Pres : Boolean;
begin
    loop
        ...
        if Ready_Temp then
            Read(TR);
            Temp_Convert(TR,HS);
            Write(HS);  -- assuming write to be reliable
            Write(TR);
        end if;
        if Ready_Pres then
            Read(PR);
            Pressure_Convert(PR,PS);
            Write(PS);
            Write(PR);
        end if;
    end loop;
end Controller;

```

This solution is more reliable; unfortunately the program now spends a high proportion of its time in a 'busy loop' polling the input devices to see if they are ready. Busy-waits are, in general, unacceptably inefficient. They tie up the processor and make it difficult to impose a queue discipline on waiting requests. Moreover, programs that rely on busy waiting are difficult to design, understand or prove correct.

The major criticism that can be levelled at the sequential program is that no recognition is given to the fact that the pressure and temperature cycles are entirely independent subsystems. In a concurrent programming environment, this can be rectified by coding each system as a task.

Using operating system primitives

Consider a POSIX-like operating system which allows a new task/thread to be created and started by calling the following Ada subprogram:

```

package Operating_System_Interface is
    type Thread_ID is private;
    type Thread is access procedure; -- a pointer type

    function Create_Thread(Code : Thread) return Thread_ID;
    -- other subprograms for thread interaction
private
    type Thread_ID is ...;
end Operating_System_Interface;

```

The simple embedded system can now be implemented as follows. First, the two controller procedures are placed in a package:

```

package Processes is
    procedure Pressure_Controller;
    procedure Temp_Controller;
end Processes;

```



```

with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
package body Processes is
  procedure Temp_Controller is
    TR : Temp_Reading;
    HS : Heater_Setting;
  begin
    loop
      Read(TR);
      Temp_Convert(TR, HS);
      Write(HS);
      Write(TR);
    end loop;
  end Temp_Controller;

  procedure Pressure_Controller is
    PR : Pressure_Reading;
    PS : Pressure_Setting;
  begin
    loop
      Read(PR);
      Pressure_Convert(PR, PS);
      Write(PS);
      Write(PR);
    end loop;
  end Pressure_Controller;
end Processes;

```

Now the Controller procedure can be given:

```

with Operating_System_Interface; use Operating_System_Interface;
with Processes; use Processes;
procedure Controller is
  Tc, Pc: Thread_Id;
begin
  -- create the threads
  -- 'Access returns a pointer to the procedure
  Tc := Create_Thread(Temp_Controller'Access);
  Pc := Create_Thread(Pressure_Controller'Access);
end Controller;

```

Procedures Temp_Controller and Pressure_Controller execute concurrently and each contains an indefinite loop within which the control cycle is defined. While one thread is suspended waiting for a read, the other may be executing; if they are both suspended a busy loop is not executed.

Although this solution does have advantages over the sequential solution, the lack of language support for expressing concurrency means that the program can become difficult to write and maintain. For the simple example given above, the added complexity is manageable. However, for large systems with many concurrent tasks and potentially complex interactions between them, having a procedural interface obscures the structure of the program. For example, it is not obvious which procedures are really procedures or which ones are intended to be concurrent activities.

Using a concurrent programming language

In a concurrent programming language, concurrent activities can be identified explicitly in the code:

```
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Controller is
  task Temp_Controller;
  task Pressure_Controller;

  task body Temp_Controller is
    TR : Temp_Reading; HS : Heater_Setting;
  begin
    loop
      Read(TR);
      Temp_Convert(TR,HS);
      Write(HS);
      Write(TR);
    end loop;
  end Temp_Controller;

  task body Pressure_Controller is
    PR : Pressure_Reading; PS : Pressure_Setting;
  begin
    loop
      Read(PR);
      Pressure_Convert(PR,PS);
      Write(PS);
      Write(PR);
    end loop;
  end Pressure_Controller;

begin
  null;    -- Temp_Controller and Pressure_Controller
          -- have started their executions
end Controller;
```

The logic of the application is now reflected in the code; the inherent parallelism of the domain is represented by concurrently executing tasks in the program.

Although an improvement, one major problem remains with this two-task solution. Both Temp_Controller and Pressure_Controller send data to the console, but the console is a resource that can only sensibly be accessed by one task at a time. In Figure 4.5, control over the console was given to a third entity (S) which will need a representation in the program – Screen_Controller. This entity may be a server or a protected resource (depending on the complete definition of the required behaviour of Screen_Controller). This has transposed the problem from one of concurrent access to a passive resource to one of inter-task communication, or at least communication between a task and some other concurrency primitive. It is necessary for tasks Temp_Controller and Pressure_Controller to pass data to Screen_Controller. Moreover, Screen_Controller must ensure that it deals with only one request at a time. These requirements and difficulties are of primary

importance in the design of concurrent programming languages, and are considered in the following chapters.

The concurrent version of this simple embedded system has a structure that is common in many control systems. Each task is iterative, it gets its data at the start of each iteration, it processes the data and it writes its results at the end of each iteration. This example will be revisited throughout this book as various aspects of real-time systems are discussed.

4.9 Language-supported versus operating-system-supported concurrency

Although this book is focusing on concurrent real-time languages, it is clear that an alternative approach is to use a sequential language, like C, and a real-time operating system (such as one that conforms to the POSIX API).

There has been a long debate among programmers, language designers and operating system designers as to whether it is appropriate to provide support for concurrency in a language or whether this should be provided by the operating system only. Arguments in favour of including concurrency in the programming languages include the following.

- (1) It leads to more readable and maintainable programs.
- (2) There are many different types of operating system; defining the concurrency in the language makes the program more portable.
- (3) An embedded computer may not have any resident operating system available.

These arguments were clearly the ones which held the most weight with the designers of Ada and Java. Arguments against concurrency in a language include the following.

- (1) Different languages have different models of concurrency; it is easier to compose programs from different languages if they all use the same operating system model of concurrency.
- (2) It may be difficult to implement a language's model of concurrency efficiently on top of an operating system's model.
- (3) Operating system API standards, such as POSIX, have emerged and therefore programs are more portable.

The need to support multiple languages was one of the main reasons why the civil aircraft industry when developing its Integrated Modular Avionics programme opted for a standard applications–kernel interface (called APEX) supporting concurrency rather than adopting the Ada model of concurrency (ARINC AEE Committee, 1999). However, it should be noted that certain compiler optimizations may lead to race conditions if the compiler does not take into account potential concurrent execution of the program (Buhr, 1995; Boehm, 2005). This is particularly true for multiprocessor systems. As a consequence of this, even C++ is adding language-defined support for multithreading into the next version of its international standard.

Summary

The application domains of most real-time systems are inherently parallel. It follows that the inclusion of the notion of task/thread within a real-time programming language makes an enormous difference to the expressive power and ease of use of the language. These factors in turn contribute significantly to reducing the software construction costs whilst improving the reliability of the final system.

Without concurrency, the software must be constructed as a single control loop. The structure of this loop cannot retain the logical distinction between system components. It is particularly difficult to give task-oriented timing and reliability requirements without the notion of a task being visible in the code.

The use of a concurrent programming language is not, however, without its costs. In particular, it becomes necessary to use a run-time support system (or operating system) to manage the execution of the system tasks.

The behaviour of a task is best described in terms of states. In this chapter, the following states are discussed:

- non-existing
- created
- initialized
- executable
- waiting dependent termination
- waiting child initialization
- terminated.

Within concurrent programming languages, there are a number of variations in the task model adopted. These variations can be analysed under six headings.

- (1) **structure** – static or dynamic task model;
- (2) **level** – top-level tasks only (flat) or multilevel (nested);
- (3) **initialization** – with or without parameter passing;
- (4) **granularity** – fine or coarse grain;
- (5) **termination** –
 - natural
 - suicide
 - aborted
 - untrapped error
 - never
 - when no longer needed;
- (6) **representation** – fork/join, cobegin, explicit task declarations.

Ada and Java provide a dynamic model with support for nested tasks and a range of termination options. C/Real-Time POSIX allows dynamic threads to be created with a flat structure; threads must explicitly terminate or be killed.

Although Ada, Real-Time Java and C/Real-Time POSIX all support multi-processor implementations, they do not provide mechanisms to set the processor affinity of a task.

Further reading

- Ben-Ari, M. (2005) *Principles of Concurrent and Distributed Programming*. New York: Prentice Hall.
- Burns, A. and Wellings, A. J. (2007) *Concurrent and Real-time Programming in Ada*. Cambridge: Cambridge University Press.
- Butenhof, D. R. (1997) *Programming With POSIX Threads*. Reading, MA: Addison-Wesley.
- Goetz, B. (2006) *Java: Concurrency in Practice*. Reading, MA: Addison-Wesley.
- Hyde, P. (1999) *Java Thread Programming*. Indianapolis, IN: Sams Publishing.
- Lea, D. (1999) *Concurrent Programming in Java: Design Principles and Patterns*. Reading, MA: Addison-Wesley.
- Oaks, A. and Wong, H. (1997) *Java Threads*. Sebastopol, CA: O'Reilly.
- Nichols, B., Buttler, D. and Farrell, J. (1996) *POSIX Threads Programming*. Sebastopol, CA: O'Reilly.
- Wellings, A. J. (2004) *Concurrent and Real-Time Programming in Java*. Chichester: Wiley.

Exercises

- 4.1 A particular operating system has a system call (*RunConcurrently*) which takes an unconstrained array. Each element of the array is a pointer to a parameterless procedure. The system call executes all the procedures concurrently and returns when all have terminated. Show how this system call can be implemented in Ada using the Ada tasking facilities. Assume that the operating system and the application run in the same address space.
- 4.2 Write Ada code to create an array of tasks where each task has a parameter which indicates its position in the array.
- 4.3 Show how a `cobegin` can be implemented in Ada.
- 4.4 Can the `fork` and `join` method of task creation be implemented in Ada without using intertask communication?
- 4.5 How many POSIX processes are created with the following procedure?


```
for(i=0; i<=10;i++) {
    fork();
}
```
- 4.6 Rewrite the simple embedded system illustrated in Section 4.8 in Java and C/Real-Time POSIX.
- 4.7 If a multithread process executes a POSIX-like `fork` system call, how many threads should the created process contain?

- 4.8** Show, using concurrent tasks, the structure of a program to control access to a simple car park. Assume that the car park has a single entrance and a single exit barrier, and a full sign.
- 4.9** Explain, with the help of the following program, the interactions between Ada's rules for task termination and its exception propagation model. Include a consideration of the program's behaviour (output) for initial values of the variable C of 2, 1 and 0.

```

with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
  task A;

  task body A is
    C : Positive := Some_Integer_Value;
    procedure P(D : Integer) is
      task T;
      A : Integer;
      task body T is
        begin
          delay 10.0;
          Put("T Finished"); New_Line;
        end T;
      begin
        Put("P Started"); New_Line;
        A := 42/D;
        Put("P Finished"); New_Line;
      end P;

      begin
        Put("A Started"); New_Line;
        P(C-1);
        Put("A Finished"); New_Line;
      end A;

    begin
      Put("Main Procedure Started"); New_Line;
    exception
      when others =>
        Put("Main Procedure Failed"); New_Line;
    end Main;

```

- 4.10** For every task in the following Ada program indicate its parent and guardian (master) and if appropriate its children and dependants. Also indicate the dependants of the Main and Hierarchy procedures.

```

procedure Main is
  procedure Hierarchy is
    task A;
    task type B;

    type Pb is access B;
    Pointerb : Pb;

    task body A is
      task C;

```



```

    task D;
    task body C is
    begin
        -- sequence of statements including
        Pointerb := new B;
    end C;
    task body D is
        Another_Pointerb : Pb;
    begin
        -- sequence of statements including
        Another_Pointerb := new B;
    end D;
begin
    -- sequence of statements
end A;

    task body B is
    begin
        -- sequence of statements
    end B;

begin
    -- sequence of statements
end Hierarchy;

begin
    -- sequence of statements
end Main;

```

4.11 To what extent can Figure 4.2 be used to represent the state transition diagram of (a) C/Real-Time POSIX pthreads and (b) Java threads?

4.12 Given the following:

```

public class Calculate implements Runnable
{
    public void run()
    {
        /* long calculation */
    }
}

```

Calculate MyCalculation = new Calculate();

what is the difference between:

```
MyCalculation.run();
```

and

```
new Thread(MyCalculation).start();
```

4.13 Explain how a Java thread can be protected against being destroyed by an arbitrary thread.

Chapter 5

Shared variable-based synchronization and communication

5.1	Mutual exclusion and condition synchronization	5.8	Protected objects in Ada
5.2	Busy waiting	5.9	Synchronized methods in Java
5.3	Suspend and resume	5.10	Shared memory multiprocessors
5.4	Semaphores	5.11	Simple embedded system revisited
5.5	Conditional critical regions		Summary
5.6	Monitors		Further reading
5.7	Mutexes and condition variables in C/Real-Time POSIX		Exercises

The major difficulties associated with concurrent programming arise from task interactions. Rarely are tasks as independent of one another as they were in the simple example at the end of Chapter 4. The correct behaviour of a concurrent program is critically dependent on synchronization and communication between tasks. In its widest sense, synchronization is the satisfaction of constraints on the interleaving of the actions of different tasks (for example, a particular action by one task only occurring after a specific action by another task). The term is also used in the narrower sense of bringing two tasks simultaneously into predefined states. Communication is the passing of information from one task to another. The two concepts are linked, since some forms of communication require synchronization, and synchronization can be considered as contentless communication.

Inter-task communication is usually based upon either **shared variables** or **message passing**. Shared variables are objects to which more than one task has access; communication can therefore proceed by each task referencing these variables when appropriate. Message passing involves the explicit exchange of data between two tasks by means of a message that passes from one task to another via some agency. Note that the choice between shared variables and message passing is one for the language or operating systems designers; it does not imply that any particular implementation method should be used. Shared variables are easy to support if there is shared memory between the tasks and, hence, they are an ideal mechanism for communication between tasks in a shared memory multiprocessor system. However, they can

still be used even if the hardware incorporates a communication medium. Similarly, a message-passing primitive is an ideal abstraction for a distributed system where there is potentially no shared physical memory but, again, it can also be supported via shared memory. Furthermore, an application can arguably be programmed in either style and obtain the same functionality (Lauer and Needham, 1978).

This chapter will concentrate on shared variable-based communication and synchronization primitives. In particular, busy waiting, semaphores, conditional critical regions, monitors, protected types and synchronized methods are discussed. The impact of shared memory multiprocessors is also considered. Message-based synchronization and communication are discussed in Chapter 6.

5.1 Mutual exclusion and condition synchronization

Although shared variables appear to be a straightforward way of passing information between tasks, their unrestricted use is unreliable and unsafe due to multiple update problems. Consider two tasks updating a shared variable, X , with the assignment:

$$X := X + 1$$

On most hardware this will not be executed as an **indivisible** (atomic) operation, but will be implemented in three distinct instructions:

- (1) load the value of X into some register (or to the top of the stack);
- (2) increment the value in the register by 1; and
- (3) store the value in the register back to X .

As the three operations are not indivisible, two tasks simultaneously updating the variable could follow an interleaving that would produce an incorrect result. For example, if X was originally 5, the two tasks could each load 5 into their registers, increment and then store 6.

A sequence of statements that must appear to be executed indivisibly is called a **critical section**. The synchronization required to protect a critical section is known as **mutual exclusion**. Atomicity, although absent from the assignment operation, is assumed to be present at the memory level. Thus, if one task is executing $X := 5$, simultaneously with another executing $X := 6$, the result will be either 5 or 6 (not some other value). If this were not true, it would be difficult to reason about concurrent programs or implement higher levels of atomicity, such as mutual exclusion synchronization. Clearly, however, if two tasks are updating a structured object, this atomicity will only apply at the single word element level.

The mutual exclusion problem itself was first described by Dijkstra (1965). It lies at the heart of most concurrent task synchronizations and is of great theoretical as well as practical interest. Mutual exclusion is not, however, the only synchronization of importance; indeed, if two tasks do not share variables then there is no need for mutual exclusion. Condition synchronization is another significant requirement and is needed

when a task wishes to perform an operation that can only sensibly, or safely, be performed if another task has itself taken some action or is in some defined state.

An example of condition synchronization comes with the use of buffers. Two tasks that exchange data may perform better if communication is not direct but via a buffer. This has the advantage of de-coupling the tasks and allows for small fluctuations in the speed at which the two tasks are working. For example, an input task may receive data in bursts that must be buffered for the appropriate user task. The use of a buffer to link two tasks is common in concurrent programs and is known as a **producer-consumer** system.

Two condition synchronizations are necessary if a finite (bounded) buffer is used. Firstly, the producer task must not attempt to deposit data into the buffer if the buffer is full. Secondly, the consumer task cannot be allowed to extract objects from the buffer if the buffer is empty. Moreover, if simultaneous deposits or extractions are possible, mutual exclusion must be ensured so that two producers, for example, do not corrupt the 'next free slot' pointer of the buffer.

The implementation of any form of synchronization implies that tasks must at times be held back until it is appropriate for them to proceed. In Section 5.2, mutual exclusion and condition synchronization will be programmed (in pseudo code with explicit task declaration) using **busy-wait** loops and **flags**. From this analysis, it should be clear that further primitives are needed to ease the coding of algorithms that require synchronization.

5.2 Busy waiting

One way to implement synchronization is to have tasks set and check shared variables that are acting as flags. This approach works reasonably well for implementing condition synchronization, but no simple method for mutual exclusion exists. To signal a condition, a task sets the value of a flag; to wait for this condition, another task checks this flag and proceeds only when the appropriate value is read.

```
task P1;  -- pseudo code for waiting task
...
while flag = down do
    null
end;
...
end P1;

task P2;  -- signalling task
...
flag := up;
...
end P2;
```

If the condition is not yet set (that is, flag is still down) then P1 has no choice but to loop round and recheck the flag. This is **busy waiting**; also known as **spinning** (with the flag variables called **spin locks**).

Busy-wait algorithms are in general inefficient; they involve tasks using up processing cycles when they cannot perform useful work. Even on a multiprocessor system,

they can give rise to excessive traffic on the memory bus or network (if distributed). Moreover, it is not possible to impose queuing disciplines easily if there is more than one task waiting on a condition (that is, checking the value of a flag). More seriously, they can leave to **livelock**. This is an error condition where tasks get stuck in their busy-wait loops and are unable to make progress.

Mutual exclusion presents even more difficulties as the algorithms required are more complex. Consider two tasks (P1 and P2 again) that have mutual critical sections. In order to protect access to these critical sections, it can be assumed that each task executes an entry protocol before the critical section and an exit protocol afterwards. Each task can therefore be considered to have the following form.

```
task P; -- pseudo code
  loop
    entry protocol
    critical section
    exit protocol
    non-critical section
  end
end P;
```

An algorithm is presented below that provides mutual exclusion and absence of livelock. It was first presented by Peterson (1981). The approach of Peterson is to have two flags (flag1 and flag2) that are manipulated by the task that 'owns' them and a turn variable that is only used if there is contention for entry to the critical sections.

```
task P1; -- pseudo code
  loop
    flag1:= up;      -- announce intent to enter
    turn:= 2;        -- give priority to other task
    while flag2 = up and turn = 2 do
      null;
    end;
    <critical section>
    flag1:= down;
    <non-critical section>
  end
end P1;

task P2;
  loop
    flag2:= up;      -- announce intent to enter
    turn:= 1;        -- give priority to other task
    while flag1 = up and turn = 1 do
      null;
    end;
    <critical section>
    flag2:= down;
    <non-critical section>
  end
end P2;
```


If only one task wishes to enter its critical section then the other task's flag will be down and entry will be immediate. However, if both flags have been raised then the value of turn becomes significant. Let us say that it has the initial value 1; then there are four possible interleavings, depending on the order in which each task assigns a value to turn and then checks its value in the while statement:

First Possibility -- P1 first then P2

P1 sets turn to 2

P1 checks turn and enters busy loop

P2 sets turn to 1 (turn will now stay with that value)

P2 checks turn and enters busy loop

P1 loops around rechecks turn and enters critical section

Second Possibility -- P2 first then P1

P2 sets turn to 1

P2 checks turn and enters busy loop

P1 sets turn to 2 (turn will now stay with that value)

P1 checks turn and enters busy loop

P2 loops around rechecks turn and enters critical section

Third Possibility -- interleaved P1 and P2

P1 sets turn to 2

P2 sets turn to 1 (turn will stay with this value)

P2 enters busy loop

P1 enters critical section

Fourth Possibility -- interleaved P2 and P1

P2 sets turn to 1

P1 sets turn to 2 (turn will stay with this value)

P1 enters busy loop

P2 enters critical section

All four possibilities lead to one task in its critical section and one task in a busy loop.

In general, although a single interleaving can only illustrate the failure of a system to meet its specification, it is not possible to show easily that all possible interleavings lead to compliance with the specification. Normally, proof methods (including model checking) are needed to show such compliance.

Interestingly, the above algorithm is fair in the sense that if there is contention for access (to their critical sections) and, say, P1 was successful (via either the first or third possible interleaving) then P2 is bound to enter next. When P1 exits its critical section, it lowers flag1. This could let P2 into its critical section, but even if it does not (because P2 was not actually executing at that time) then P1 would proceed, enter and leave its non-critical section, raise flag1, set turn to 2 and then be placed in a busy loop. There it would remain until P2 had entered and left its critical section and reset flag2 as its exit protocol.

In terms of reliability, the failure of a task in its non-critical section will not affect the other task. This is not the case with failure in the protocols or critical section. Here, premature termination of a task would lead to livelock difficulties for the remaining program.

This discussion has been given at length to illustrate the difficulties of implementing synchronization between tasks with only shared variables and no additional primitives

other than those found in sequential languages. These difficulties can be summarized as follows.

- Protocols that use busy loops are difficult to design, understand and prove correct. (The reader might like to consider generalizing Peterson's algorithm for n tasks.)
- Testing programs may not examine rare interleavings that break mutual exclusion or lead to livelock.
- Busy-wait loops are inefficient.
- An unreliable (rogue) task that misuses shared variables will corrupt the entire system.

No concurrent programming language relies entirely on busy waiting and shared variables; other methods and primitives have been introduced. For shared-variable systems, semaphores and monitors are the most significant constructs and are described in Sections 5.4 and 5.6.

5.3 Suspend and resume

One of the problems with busy-wait loops is that they waste valuable processor time. An alternative approach is to suspend (that is, remove from the set of runnable tasks) the calling task if the condition for which it is waiting does not hold. Consider, for example, simple condition synchronization using a flag. One task sets the flag, and another task waits until the flag is set and then clears it. A simple suspend and resume mechanism could be used as follows:

```
task P1;  -- pseudo code for waiting task
...
if flag = down do
    suspend;
end;
flag := down;
...
end P1;

task P2;  -- signalling task
...
flag := up;
resume P1; -- has no effect, if P1 is not suspended
...
end P2;
```

An early version of the Java Thread class provided the following methods in support of this approach.

```
public final void suspend();
// throws SecurityException;
public final void resume();
// throws SecurityException;
```


Thus the above example would be represented in Java.

```

boolean flag;
final boolean up = true;
final boolean down = false;

class FirstThread extends Thread {

    public void run() {
        ...
        if(flag == down) {
            suspend();
        };
        flag = down;
        ...
    }
}

class SecondThread extends Thread { // T2

    FirstThread T1;

    public SecondThread(FirstThread T) {
        super();
        T1 = T;
    }

    public void run() {
        ...
        flag = up;
        T1.resume();
        ...
    }
}

```

Unfortunately, this approach suffers from what is called a **data race condition**.

A data race condition is a fault in the design of the interactions between two or more tasks whereby the result is unexpected and critically dependent on the sequence or timing of accesses to shared data.

In this case, thread T1 could test the flag, and then the underlying run-time support system (or operating system) could decide to preempt it and run T2. T2 sets the flag and resumes T1. T1 is, of course, not suspended, so the resume has no effect. Now, when T1 next runs, it thinks the flag is down and therefore suspends itself.

The reason for this problem is that the flag is a shared resource which is being tested and an action is being taken which depends on its status (the thread is suspending itself). This testing and suspending is not an atomic operation, and therefore interference can occur from other threads. It is for this reason that the most recent version of Java has made these methods obsolete.

There are several well-known solutions to this race condition problem, all of which provide a form of **two-stage suspend** operation. P1 essentially has to announce that it

Program 5.1 Synchronous task control.

```

package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True(S : in out Suspension_Object);
  procedure Set_False(S : in out Suspension_Object);
  function Current_State(S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True(S: in out Suspension_Object);
    -- raises Program_Error if more than one task tries
    -- to suspend on S at once.
private
  -- not specified by the language
end Ada.Synchronous_Task_Control;

```

is planning to suspend in the near future; any resume operation which finds that P1 is not suspended will have a deferred effect. When P1 does suspend, it will immediately be resumed; that is, the suspend operation itself will have no effect.

Although suspend and resume is a low-level facility, which can be error-prone in its use, it is an efficient mechanism which can be used to construct higher-level synchronization primitives. For this reason, Ada provides, as part of its Real-Time Annex, a safe version of this mechanism. It is based around the concept of a **suspension** object, which can hold the value `True` or `False`. Program 5.1 gives the package specification.

All four subprograms defined by the package are atomic with respect to each other. On return from the `Suspend_Until_True` procedure, the referenced suspension object is reset to `False`.

The simple condition synchronization problem, given earlier in this section, can, therefore, be easily solved.

```

with Ada.Synchronous_Task_Control;
use Ada.Synchronous_Task_Control;

...
Flag : Suspension_Object;
...
task body P1 is
begin
  ...
  Suspend_Until_True(Flag);
  ...
end P1;

task body P2 is
begin
  ...
  Set_True(Flag);
  ...
end P2;

```

Suspension objects behave in much the same way as binary semaphores, which are discussed in Section 5.4.4.

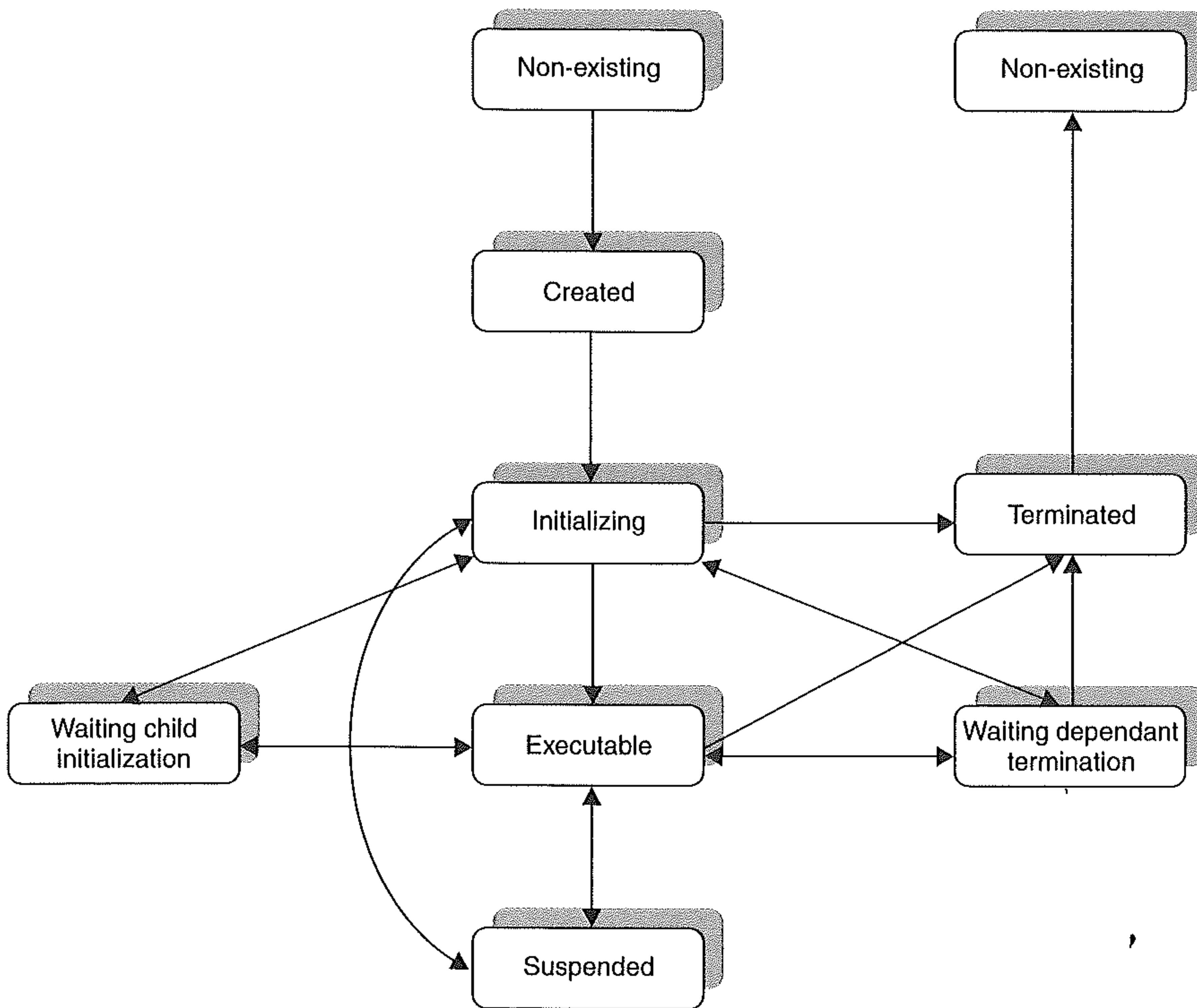


Figure 5.1 State diagram for a task.

Although `suspend` and `resume` are useful low-level primitives, no operating system or language relies solely on these mechanisms for mutual exclusion and condition synchronization. If present, they clearly introduce a new state into the state transition diagram introduced in Chapter 4. The general state diagram for a task, therefore, is extended in Figure 5.1.

5.4 Semaphores

Semaphores are a simple mechanism for programming mutual exclusion and condition synchronization. They were originally designed by Dijkstra (1968) and have the following two benefits.

- (1) They simplify the protocols for synchronization.
- (2) They remove the need for busy-wait loops.

A **semaphore** is a non-negative integer variable that, apart from initialization, can only be acted upon by two procedures. These procedures are called `wait` and `signal` in this book. The semantics of `wait` and `signal` are as follows.

- (1) `wait(S)` – If the value of the semaphore, `S`, is greater than zero then decrement its value by one; otherwise delay the task until `S` is greater than zero (and then decrement its value).
- (2) `signal(S)` – Increment the value of the semaphore, `S`, by one.

General semaphores are often called **counting semaphores**, as their operations increment and decrement an integer count. The additional important property of `wait` and `signal` is that their actions are atomic (indivisible). Two tasks, both executing `wait` operations on the same semaphore, cannot interfere with each other. Moreover, a task cannot fail during the execution of a semaphore operation.

Condition synchronization and mutual exclusion can be programmed easily with semaphores. First, consider condition synchronization:

```
-- pseudo code for condition synchronization
consyn : semaphore; -- initially 0
task P1; -- waiting task
    ...
    wait(consyn);
    ...
end P1;

task P2; -- signalling task
    ...
    signal(consyn);
    ...
end P2;
```

When `P1` executes the `wait` on a 0 semaphore, it will be delayed until `P2` executes the `signal`. This will set `consyn` to 1 and hence the `wait` can now succeed; `P1` will continue and `consyn` will be decremented to 0. Note that if `P2` executes the `signal` first, the semaphore will be set to 1, so `P1` will not be delayed by the action of the `wait`.

Mutual exclusion is similarly straightforward:

```
-- pseudo code for mutual exclusion

mutex : semaphore; -- initially 1
task P1;
    loop
        wait(mutex);
        <critical section>
        signal(mutex);
        <non-critical section>
    end
end P1;

task P2;
    loop
        wait (mutex);
        <critical section>
        signal (mutex);
        <non-critical section>
    end
end P2;
```


If P1 and P2 are in contention then they will execute their `wait` statements simultaneously. However, as `wait` is atomic, one task will complete execution of this statement before the other begins. One task will execute a `wait(mutex)` with `mutex=1`, which will allow the task to proceed into its critical section and set `mutex` to 0; the other task will execute `wait(mutex)` with `mutex=0`, and be delayed. Once the first task has exited its critical section, it will `signal(mutex)`. This will cause the semaphore to become 1 again and allow the second task to enter its critical section (and set `mutex` to 0 again).

With a `wait/signal` bracket around a section of code, the initial value of the semaphore will restrict the maximum amount of concurrent execution of the code. If the initial value is 0, no task will ever enter; if it is 1 then a single task may enter (that is, mutual exclusion); for values greater than one, the given number of concurrent executions of the code is allowed.

5.4.1 Suspended tasks

In the definition of `wait` it is clear that if the semaphore is zero then the calling task is delayed. One method of delay (busy waiting) has already been introduced and criticized. A more efficient mechanism, that of suspending the task, was introduced in Section 5.3. In fact, all synchronization primitives deal with delay by some form of suspension; the task is removed from the set of executable tasks.

When a task executes a `wait` on a zero semaphore, the RTSS (run-time support system) is invoked, the task is removed from the processor, and placed in a queue of suspended tasks (that is a queue of tasks suspended on that particular semaphore). The RTSS must then select another task to run. Eventually, if the program is correct, another task will execute a `signal` on that semaphore. As a result, the RTSS will pick out one of the suspended tasks awaiting a signal on that semaphore and make it executable again.

From these considerations, a slightly different definition of `wait` and `signal` can be given. This definition is closer to what an implementation would do:

```
-- pseudo code for wait(S)
if S > 0 then
    S := S-1;
else
    number_suspended := number_suspended + 1
    suspend_calling_task;

-- pseudo code signal(S)
if number_suspended > 0 then
    number_suspended := number_suspended - 1;
    make_one_suspended_task_executable_again;
else
    S := S+1;
end if;
```

With this definition, the increment of a semaphore immediately followed by its decrement is avoided.

Note that the above algorithm does not define the order in which tasks are released from the suspended state. Usually, they are released in a FIFO order, although arguably

with a true concurrent language, the programmer should assume a non-deterministic order (see Section 6.6). However, for a real-time programming language, the priority of the tasks has an important role to play (see Chapter 11).

5.4.2 Implementation

The above algorithm for implementing a semaphore is quite straightforward, although it involves the support of a queue mechanism. Where difficulty could arise is in the requirement for indivisibility in the execution of the `wait` and `signal` operations. Indivisibility means that once a task has started to execute one of these procedures it will continue to execute until the operation has been completed. With the aid of the RTSS, this is easily achieved; the scheduler is programmed so that it does not swap out a task while it is executing a `wait` or a `signal`; they are **non-preemptible** operations.

Unfortunately, the RTSS is not always in full control of scheduling events. Although all internal actions are under its influence, external actions happen asynchronously and could disturb the atomicity of the semaphore operations. To prohibit this, the RTSS will typically disable interrupts for the duration of the execution of the indivisible sequence of statements. In this way, no external events can interfere.

This disabling of interrupts is adequate for a single processor system but not for a multiprocessor one. With a shared-memory system, two parallel tasks may be executing a `wait` or `signal` (on the same semaphore) and the RTSS is powerless to prevent it. In these circumstances, a 'lock' mechanism is needed to protect access to the operations. Two such mechanisms are used.

On some processors, a 'test and set' instruction is provided. This allows a task to access a bit in the following way.

- (1) If the bit is zero then set it to one and return zero.
- (2) If the bit is one return one.

These actions are themselves indivisible. Two parallel tasks, both wishing to operate a `wait` (for example), will do a test and set operation on the same lock bit (which is initially zero). One task will succeed and set the bit to one; the other task will have returned a one and will, therefore, have to loop round and retest the lock. When the first task has completed the wait operation, it will assign the bit to zero (that is, unlock the semaphore) and the other task will proceed to execute its `wait` operation.

If no test and set instruction is available then a similar effect can be obtained by a swap instruction. Again, the lock is associated with a bit that is initially zero. A task wishing to execute a semaphore operation will swap a one with the lock bit. If it gets a zero back from the lock then it can proceed; if it gets back a one then some other task is active with the semaphore and it must retest.

As was indicated in Section 5.1, a software primitive such as a semaphore cannot conjure up mutual exclusion out of 'fresh air'. It is necessary for memory locations to exhibit the essence of mutual exclusion in order for higher-level structures to be built. Similarly, although busy-wait loops are removed from the programmer's domain by the use of semaphores, it may be necessary to use busy waits (as above) to implement the wait and signal operations. *It should be noted, however, that the latter use of busy-waits is only short-lived (the time it takes to execute a wait or signal operation), whereas their*

use for delaying access to the program's critical sections could involve many seconds of looping.

5.4.3 Liveness provision

In Section 5.2, the error condition livelock was illustrated. Unfortunately (but inevitably), the use of synchronization primitives introduces other error conditions. **Deadlock** is the most serious such condition and entails a set of tasks being in a state from which it is impossible for any of them to proceed. This is similar to livelock but the tasks are suspended. To illustrate this condition, consider two tasks P1 and P2 wishing to gain access to two non-concurrent resources (that is, resources that can only be accessed by one task at a time) that are protected by two semaphores S1 and S2. If both tasks access the resource in the same order then no problem arises:

P1	P2
wait (S1);	wait (S1);
wait (S2);	wait (S2);
.	.
.	.
.	.
signal (S2);	signal (S2);
signal (S1);	signal (S1);

The first task to execute the `wait` on S1 successfully will also successfully undertake the `wait` on S2 and subsequently `signal` the two semaphores and allow the other task in. A problem occurs, however, if one of the tasks wishes to use the resources in the reverse order, for example:

P1	P2
wait (S1);	wait (S2);
wait (S2);	wait (S1);
.	.
.	.
.	.
signal (S2);	signal (S1);
signal (S1);	signal (S2);

In this case, an interleaving could allow P1 and P2 to execute successfully the `wait` on S1 and S2, respectively, but then inevitably both tasks will be suspended waiting on the other semaphore which is now zero.

It is in the nature of an interdependent concurrent program that usually once a subset of the tasks becomes deadlocked all the other tasks will eventually become part of the deadlocked set.

The testing of software rarely removes other than the most obvious deadlocks; they can occur infrequently but with devastating results. This error is not isolated to the use of semaphores and is possible in all concurrent programming languages. The design of languages that prohibit the programming of deadlocks is a desirable, but not yet attainable, goal. Issues relating to deadlock avoidance, detection and recovery will be considered in Chapters 8 and 11.

Indefinite postponement (sometimes called **lockout** or **starvation**) is a less severe error condition whereby a task that wishes to gain access to a resource, via a critical section, is never allowed to do so because there are always other tasks gaining access before it. With a semaphore system, a task may remain indefinitely suspended (that is, queued on the semaphore) due to the way the RTSS picks tasks from this queue when a signal arrives. Even if the delay is not in fact indefinite, but merely open ended (indeterminate), this may give rise to an error in a real-time system.

If a task is free from livelocks, deadlocks and indefinite postponements then it is said to possess **liveness**. Informally, the liveness property implies that if a task wishes to perform some action then it will, eventually, be allowed to do so. In particular, if a task requests access to a critical section it will gain access within a finite time.

5.4.4 Binary and quantity semaphores

The definition of a (general) semaphore is a non-negative integer; by implication its actual value can rise to any supported positive number. However, in all the examples given so far in this chapter (that is, for condition synchronization and mutual exclusion), only the values 0 and 1 have been used. A simple form of semaphore, known as a **binary semaphore**, can be implemented that takes only these values; that is, the signalling of a semaphore which has the value 1 has no effect – the semaphore retains the value 1. The construction of a general semaphore from two binary semaphores and an integer can then be achieved, if the general form is required.

Another variation on the normal definition of a semaphore is the **quantity semaphore**. With this structure, the amount to be decremented by the `wait` (and incremented by the `signal`) is not fixed as 1, but is given as a parameter to the procedures:

```
wait(S, i) :-   if S >= i then
                  S := S-i
                else
                  delay
                  S := S-i
signal(S, i) :- S := S+i
```

5.4.5 Example semaphore programs in Ada

Algol-68 was the first language to introduce semaphores. It provided a type `sema` that was manipulated by the operators `up` and `down`. To illustrate some simple programs that use semaphores, an abstract data type for semaphores, in Ada, will be used.

```
package Semaphore_Package is
  type Semaphore(Initial : Natural := 1) is limited private;
  procedure Wait (S : in out Semaphore);
  procedure Signal (S : in out Semaphore);
private
  type Semaphore is ...
end Semaphore_Package;
```

Ada does not directly support semaphores, but the `Wait` and `Signal` procedures can; however, be constructed from the Ada synchronization primitives; these have not yet

been discussed, so the full definition of the type semaphore and the body of the package will not be given here (see Section 5.8). The essence of abstract data types is, however, that they can be used without knowledge of their implementation.

The first example is the producer/consumer system that uses a bounded buffer to pass integers between the two tasks:

```

procedure Main is
  package Buffer is
    procedure Append (I : Integer);
    procedure Take (I : out Integer);
  end Buffer;
  task Producer;
  task Consumer;

  package body Buffer is separate;
  use Buffer;

  task body Producer is
    Item : Integer;
  begin
    loop
      -- produce item
      Append (Item);
    end loop;
  end Producer;

  task body Consumer is
    Item : Integer;
  begin
    loop
      Take (Item);
      -- consume item
    end loop;
  end Consumer;
begin
  null;
end Main;

```

The buffer itself must protect against concurrent access, appending to a full buffer and taking from an empty one. This it does by the use of three semaphores:

```

with Semaphore_Package; use Semaphore_Package;
separate (Main)
package body Buffer is
  Size : constant Natural := 32;
  type Buffer_Range is mod Size;
  Buf : array (Buffer_Range) of Integer;
  Top, Base : Buffer_Range := 0;

  Mutex : Semaphore; -- default is 1
  Item_Available : Semaphore(0);
  Space_Available : Semaphore(Initial => Size);

  procedure Append (I : Integer) is
  begin

```



```

    Wait(Space_Available);
    Wait(Mutex);
    Buf(Top) := I;
    Top := Top + 1;
    Signal(Mutex);
    Signal(Item_Available);
end Append;

procedure Take (I : out Integer) is
begin
    Wait(Item_Available);
    Wait(Mutex);
    I := Buf(Base);
    Base := Base + 1;
    Signal(Mutex);
    Signal(Space_Available);
end Take;
end Buffer;

```

The initial values of the three semaphores are different. `Mutex` is an ordinary mutual exclusion semaphore and is given the default initial value of 1; `Item_Available` protects against taking from an empty buffer and has the initial value 0; and `Space_Available` (initially `Size`) is used to prevent `Append` operations to a full buffer.

When the program starts, any consumer task that calls `Take` will be suspended on `Wait(Item_Available)`; only after a producer task has called `Append`, and in doing so `Signal(Item_Available)`, will the consumer task continue.

5.4.6 Semaphore programming using Java

Although the Java language supports a monitor-like communication and synchronization model (see Section 5.9), the Java platform provides several standard packages that support concurrency utilities. One of these provides general-purpose classes to support different synchronization approaches. Semaphores are included in this package.

5.4.7 Semaphore programming using C/Real-Time POSIX

Although few modern programming languages support semaphores directly, many operating systems do. The POSIX API, for example, provides counting semaphores to enable processes running in separate address spaces (or threads within the same address space) to synchronize and communicate using shared memory. Note, however, that it is more efficient to use mutexes and condition variables to synchronize and communicate in the same address space – see Section 5.7. Program 5.2 defines the C/Real-Time POSIX interface for semaphores (functions for naming a semaphore by a character string are also provided but have been omitted here). The standard semaphore operations *initialize*, *wait* and *signal* are called `sem_init`, `sem_wait` and `sem_post` in C/Real-Time POSIX. A non-blocking wait (`sem_trywait`) and a timed-version (`sem_timedwait`) are also provided, as is a routine to determine the current value of a semaphore (`sem_getvalue`).

Consider an example of a resource controller which appears in many forms in real-time programs. For simplicity, the example will use threads rather than processes. Two functions are provided: `allocate` and `deallocate`; each takes a parameter

Program 5.2 The C/Real-Time POSIX interface to semaphores.

```

#include <time.h>
typedef ... sem_t;
int sem_init(sem_t *sem_location, int pshared, unsigned int value);
    /* initializes the semaphore at location sem_location to value */
    /* if pshared is 1, the semaphore can be used between processes */
    /*    or threads */
    /* if pshared is 0, the semaphore can only be used between threads */
    /* of the same process */

int sem_destroy(sem_t *sem_location);
    /* remove the unnamed semaphore at location sem_location */

int sem_wait(sem_t *sem_location);
    /* a standard wait operation on a semaphore */

int sem_trywait(sem_t *sem_location);
    /* attempts to decrement the semaphore */
    /* returns -1 if the call might block the calling process */

int sem_timedwait(sem_t *sem, const struct timespec *abstime);
    /* returns -1 if the semaphore could not be locked */
    /* by abstime */

int sem_post(sem_t *sem_location);
    /* a standard signal operation on a semaphore */

int sem_getvalue(sem_t *sem_location, int *value);
    /* gets the current value of the semaphore to a location */
    /* pointed at by value; negative value indicates the number */
    /* of threads waiting */

/* All the above functions return 0 if successful, otherwise -1. */
/* When an error condition is returned by any of the above */
/* functions, a shared variable errno contains the reason for */
/* the error */

```

which indicates a priority level associated with the request. It is assumed that the calling thread deallocates the resource at the same priority with which it requested allocation. For ease of presentation, the example does not consider how the resource itself is transferred. Moreover, the solution does not protect itself against race conditions (see Exercise 5.21).

```

#include <semaphore.h>

typedef enum {high, medium, low} priority_t;
typedef enum {false, true} boolean;

sem_t mutex;    /* used for mutual exclusive
                  access to waiting and busy */
sem_t cond[3]; /* used for condition synchronization */

```

```

int waiting;    /* count of number of threads
                  waiting at a priority level */
int busy; /* indicates whether the resource is in use*/

void allocate(priority_t P)
{
    SEM_WAIT(&mutex); /* lock mutex */
    if(busy) {
        SEM_POST(&mutex); /* release mutex */
        SEM_WAIT(&cond[P]); /* wait at correct priority level */
        /* resource has been allocated */
    }
    busy = true;
    SEM_POST(&mutex); /* release mutex */
}

```

A single semaphore, `mutex`, is used to ensure that all allocation and deallocation requests are handled in mutual exclusion. Three condition synchronization semaphores, `cond[3]`, are used to queue the waiting threads at three priority levels (high, medium and low). The `allocate` function allocates the resource if it is not already in use (indicated by the `busy` flag).

The deallocation function simply signals the semaphore of the highest priority waiter.

```

int deallocate(priority_t P)
{
    SEM_WAIT(&mutex); /* lock mutex */
    if(busy) {
        busy = false;
        /* release highest priority waiting thread */
        SEM_GETVALUE(&cond[high], &waiting);
        if (waiting < 0) {
            SEM_POST(&cond[high]);
        }
        else {
            SEM_GETVALUE(&cond[medium], &waiting);
            if (waiting < 0) {
                SEM_POST(&cond[medium]);
            }
            else {
                SEM_GETVALUE(&cond[low], &waiting);
                if (waiting < 0) {
                    SEM_POST(&cond[low]);
                }
                else SEM_POST(&mutex);
                /* no one waiting, release lock */
            }
        }
        /* resource and lock passed on to */
        /* highest priority waiting thread */
        return 0;
    }
    else return -1; /* error return */
}

```


An initialization routine sets the busy flag to false and creates the four semaphores used by `allocate` and `deallocate`.

```
void initialize() {
    priority_t i;

    busy = false;
    SEM_INIT(&mutex, 0, 1);
    for (i = high; i <= low; i++) {
        SEM_INIT(&cond[i], 0, 0);
    };
}
```

Remember that, as the C binding to Real-Time POSIX uses non-zero return values to indicate an error has occurred, it is necessary to encapsulate every POSIX call in an `if` statement. This makes the code more difficult to understand (an Ada or C++ binding to POSIX would allow exceptions to be raised when errors occur). Consequently, as with the other C examples used in this book, `SYS_CALL` is used to represent a call to `sys_call` and any appropriate error recovery (see Section 3.1.1). For `SEM_INIT` this might include a retry.

A thread wishing to use the resource would make the following calls:

```
priority_t my_priority;

...
allocate(my_priority); /* wait for resource */
/* use resource */
if(deallocate(my_priority) <= 0) {
    /* cannot deallocate resource, */
    /* undertake some recovery operation */
}
```

5.4.8 Criticisms of semaphores

Although the semaphore is an elegant low-level synchronization primitive, a real-time program built only upon the use of semaphores is again error-prone. It needs just one occurrence of a semaphore to be omitted or misplaced for the entire program to collapse at run-time. Mutual exclusion may not be assured and deadlock may appear just when the software is dealing with a rare but critical event. What is required is a more structured synchronization primitive.

What the semaphore provides is a means to program mutual exclusion over a critical section. A more structured approach would give mutual exclusion directly. This is precisely what is provided for by the constructs discussed in Sections 5.5 to 5.9.

The examples shown in Section 5.4.5 showed that an abstract data type for semaphores can be constructed in Ada. However, no high-level concurrent programming language relies entirely on semaphores. They are important historically but are arguably not adequate for the real-time domain.

5.5 Conditional critical regions

Conditional critical regions (CCRs) are an attempt to overcome some of the problems associated with semaphores. A critical region is a section of code that is guaranteed to be executed in mutual exclusion. This must be compared with the concept of a critical section that should be executed under mutual exclusion (but in error may not be). Clearly, the programming of a critical section as a critical region immediately meets the requirement for mutual exclusion.

Variables that must be protected from concurrent usage are grouped together into named regions and are tagged as being resources. Processes are prohibited from entering a region in which another task is already active. Condition synchronization is provided by guards on the regions. When a task wishes to enter a critical region, it evaluates the guard (under mutual exclusion); if the guard evaluates true, it may enter, but if it is false, the task is delayed. As with semaphores, the programmer should not assume any order of access if more than one task is delayed attempting to enter the same critical region (for whatever reason).

To illustrate the use of CCRs, an outline of the bounded buffer program is given below.

```
-- pseudo code
program buffer_eg;
  type buffer_t is record
    slots      : array(1..N) of character;
    size       : integer range 0..N;
    head, tail : integer range 1..N;
  end record;

  buffer : buffer_t;

  resource buf : buffer;

  task producer;
    ...
  loop
    region buf when buffer.size < N do
      -- place char in buffer etc
    end region;
    ...
  end loop;
end,

  task consumer;
    ...
  loop
    region buf when buffer.size > 0 do
      -- take char from buffer etc
    end region
    ...
  end loop;
end;
end;
```


One potential performance problem with CCRs is that tasks must re-evaluate their guards every time a CCR naming that resource is left. A suspended task must become executable again in order to test the guard; if it is still false, it must return to the suspended state.

A version of CCRs has been implemented in Edison (Brinch-Hansen, 1981), a language intended for embedded applications, implemented on multiprocessor systems. Each processor only executes a single task so that it may continually evaluate its guards if necessary. However, this may cause excess traffic on the network.

5.6 Monitors

The main problem with conditional critical regions is that they can be dispersed throughout the program. Monitors are intended to alleviate this problem by providing more structured control regions. They also use a form of condition synchronization that is more efficient to implement.

The intended critical regions are written as procedures and are encapsulated together into a single module called a monitor. As a module, all variables that must be accessed under mutual exclusion are hidden; additionally, as a monitor, all procedure calls into the module are guaranteed to execute with mutual exclusion.

Monitors appeared as a refinement of conditional critical regions. They can be found in numerous programming languages including Modula-1, Concurrent Pascal and Mesa.

To continue, for comparison, with the bounded buffer example, a buffer monitor would have the following structure:

```
monitor buffer; -- pseudo code
  export append, take;
  -- declaration of necessary variables

  procedure append (I : integer);
    ...
  end;

  procedure take (I : integer);
    ...
  end;
begin
  -- initialization of monitor variables
end
```

With languages that support monitors, concurrent calls to append and/or take (in the above example) are serialized – by definition. No mutual exclusion semaphore needs be provided by the programmer. The languages run-time support system will implement the appropriate entry and exit protocols.

Although providing for mutual exclusion, there is still a need for condition synchronization within the monitor. In theory, semaphores could still be used, but normally a simpler synchronization primitive is introduced. In Hoare's monitors (Hoare, 1974), this primitive is called a **condition variable** and is acted upon by two operators which,

because of similarities with the semaphore structure, will again be called `wait` and `signal`. When a task issues a `wait` operation, it is blocked (suspended) and placed on a queue associated with that condition variable (this can be compared with a `wait` on a semaphore with a value of zero; however, note that a `wait` on a condition variable *always* blocks unlike a `wait` on a semaphore). A blocked task then releases its mutually exclusive hold on the monitor, allowing another task to enter. When a task executes a `signal` operation, it will release one blocked task. If no task is blocked on the specified variable then the *signal has no effect*. (Again note the contrast with `signal` on a semaphore, which always has an effect on the semaphore. Indeed, `wait` and `signal` for monitors are more akin to `suspend` and `resume` in their semantics.) The bounded buffer example can now be given in full:

```
monitor buffer; -- pseudo code
  export append, take;
  constant size = 32;
  buf : array[0...size-1] of integer;
  top, base : 0..size-1;
  SpaceAvailable, ItemAvailable : condition;
  NumberInBuffer : integer;

  procedure append (I : integer);
  begin
    if NumberInBuffer = size then
      wait(SpaceAvailable);
    buf[top] := I;
    NumberInBuffer := NumberInBuffer+1;
    top := (top+1) mod size;
    signal(ItemAvailable);
  end append;

  procedure take (var I : integer);
  begin
    if NumberInBuffer = 0 then
      wait(ItemAvailable);
    I := buf[base];
    base := (base+1) mod size;
    NumberInBuffer := NumberInBuffer-1;
    signal(SpaceAvailable);
  end take;

begin -- initialization
  NumberInBuffer := 0;
  top := 0;
  base := 0;
end;
```

If a task calls (for example) `take` when there is nothing in the buffer, then it will become suspended on `ItemAvailable`. A task appending an item will, however, `signal` this suspended task when an item does become available.

The semantics for `wait` and `signal`, given above, are not complete; as they stand, two or more tasks could become active within a monitor. This would occur following a `signal` operation in which a blocked task was freed. The freed task and the one that

freed it are then both executing inside the monitor. To prohibit this clearly undesirable activity, the semantics of `signal` must be modified. Four different approaches are used in languages.

- (1) A signal is allowed only as the last action of a task before it leaves the monitor (this is the case with the buffer example above).
- (2) A signal operation has the side-effect of executing a return statement; that is, the task is forced to leave the monitor.
- (3) A signal operation which unblocks another task has the effect of blocking itself; this task will only execute again when the monitor is free.
- (4) A signal operation which unblocks another task does not block and the freed task must compete for access to the monitor once the signalling task exits.

In case (3), which was proposed by Hoare in his original paper on monitors, the tasks that are blocked because of a signal action are placed on a 'ready queue' and are chosen, when the monitor is free, in preference to tasks blocked on entry. In case (4), it is the freed task which is placed on the 'ready queue'.

C/Real-Time POSIX, Ada and Java all support variation of the monitor approach and will be considered in detail in the Sections 5.7–5.9.

5.6.1 Nested monitor calls

A nested monitor call occurs where a monitor procedure calls a procedure defined within another monitor. This can cause problems when the nested procedure suspends on a condition variable. The mutual exclusion in the last monitor call will be relinquished by the task, due to the semantics of the wait and equivalent operations. However, mutual exclusion will not be relinquished in the monitors from which the nested call has been made. Processes that attempt to invoke procedures in these monitors will become blocked. This can have performance implications, since blockage will decrease the amount of concurrency exhibited by the system.

Various approaches to the nested monitor problem have been suggested. The most popular one, adopted by Java and C/Real-Time POSIX, is to maintain the lock. Other approaches include prohibiting nested procedure calls altogether and providing constructs which specify that certain monitor procedures may release their mutual exclusion lock during remote calls.

5.6.2 Criticisms of monitors

The monitor gives a structured and elegant solution to mutual exclusion problems such as the bounded buffer. It does not, however, deal well with condition synchronizations, resorting to low-level semaphore-like primitives. All the criticisms surrounding the use of semaphores apply equally (if not more so) to condition variables.

In addition, although monitors encapsulate all the entities concerned with a resource, and provide the important mutual exclusion, their internal structure may still be difficult to understand due to the use of condition variables.

5.7 Mutexes and condition variables in C/Real-Time POSIX

In Section 5.4.7, C/Real-Time POSIX semaphores were described as a mechanism for use between processes and between threads. If the threads extension to C/Real-Time POSIX is supported then using semaphores for communication and synchronization between threads in the same address space is expensive as well as being unstructured. **Mutexes and condition variables**, when combined, provide the functionality of a monitor but with a procedural interface. Programs 5.3 and 5.4 define the basic C interface. Program 5.3 defines the attributes associated with mutexes and condition variables. As with pthreads, each is defined by a separate object.

Program 5.3 The C/Real-Time POSIX interface to mutexes and condition variable attributes.

```
typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;
typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
    /* destroy the mutex attribute object */
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
    /* initialize a mutex attribute object */

int pthread_mutexattr_getpshared(const pthread_mutexattr_t *
    restrict attr, int *restrict pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
    int pshared);
    /* get and set the attribute that indicates that the mutex */
    /* can be used between threads in different processes */

int pthread_mutexattr_gettype(
    const pthread_mutexattr_t *restrict attr,
    int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
    /* get and set the attribute that defines the amount of */
    /* error detection that is undertaken when mutexes are used. */
    /* e.g. unlocking a unlocked mutex */

int pthread_condattr_init();
int pthread_condattr_destroy();
    /* initialize and destroy a condition attribute object */
    /* undefined behaviour if threads are waiting on the */
    /* condition variable when it is destroyed */
int pthread_condattr_getpshared();
int pthread_condattr_setpshared();
    /* get and set the attribute that indicates that the condition */
    /* can be used between threads in different processes */

...
    /* other scheduling related attributes */
```

Program 5.4 The C/Real-Time POSIX interface to mutexes and condition variables.

```

typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;
typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;

int pthread_mutex_init(pthread_mutex_t *mutex,
                      const pthread_mutexattr_t *attr);
    /* initializes a mutex with certain attributes */
int pthread_mutex_destroy(pthread_mutex_t *mutex);
    /* destroys a mutex */
    /* undefined behaviour if the mutex is locked */

int pthread_mutex_lock(pthread_mutex_t *mutex);
    /* lock the mutex; if locked already suspend calling thread */
    /* the owner of the mutex is the thread which locked it */
int pthread_mutex_trylock(pthread_mutex_t *mutex);
    /* as above, but gives an error return if mutex is already locked */
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                           const struct timespec *abstime);
    /* as for lock, but return an error if the lock cannot */
    /* be obtained by the timeout */

int pthread_mutex_unlock(pthread_mutex_t *mutex);
    /* unlocks the mutex if called by the owning thread */
    /* when successful, results in a blocked thread being released */

int pthread_cond_wait(pthread_cond_t *cond,
                     pthread_mutex_t *mutex);
    /* called by thread which owns a locked mutex */
    /* atomically blocks the calling thread on the cond variable and */
    /* releases the lock on mutex */
    /* a successful return indicates that the mutex has been locked */
int pthread_cond_timedwait(pthread_cond_t *cond,
                          pthread_mutex_t *mutex, const struct timespec *abstime);
    /* the same as pthread_cond_wait, except that a error is returned */
    /* if the timeout expires */

int pthread_cond_signal(pthread_cond_t *cond);
    /* unblocks at least one blocked thread */
    /* no effect if no threads are blocked */
    /* unblocked threads automatically contend for the associated mutex */
int pthread_cond_broadcast(pthread_cond_t *cond);
    /* unblocks all blocked threads */
    /* no effect if no threads are blocked */
    /* unblocked threads automatically contend for the associated mutex */

/* All the above functions return 0 if successful */

```

Each monitor has an associated (initialized) mutex variable, and all operations on the monitor (critical regions) are surrounded by calls to lock (`pthread_mutex_lock`) and unlock (`pthread_mutex_unlock`) the mutex.

Condition synchronization is provided by associating condition variables with the mutex. Note that, when a thread waits on a condition variable (`pthread_cond_wait`, `pthread_cond_timedwait`), its lock on the associated mutex is released. Also, when it successfully returns from the conditional wait, it again holds the lock. However, because more than one thread could be released (even by `pthread_cond_signal`), the program must again test for the condition that caused it to wait initially.

Consider the following integer bounded buffer using mutexes and condition variables. The buffer consists of a mutex, two condition variables (`buffer_not_full` and `buffer_not_empty`), a count of the number of items in the buffer, the buffer itself, and the positions of the first and last items in the buffer. The append routine locks the buffer and if the buffer is full, waits on the condition variable `buffer_not_full`. When the buffer has space, the integer data item is placed in the buffer, the mutex is unlocked and the `buffer_not_empty` signal sent. The take routine is similar in structure.

```
#include <threads.h>

#define BUFF_SIZE 10

typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t buffer_not_full;
    pthread_cond_t buffer_not_empty;
    int count, first, last;
    int buf[BUFF_SIZE];
} buffer;

int append(int item, buffer *B ) {
    PTHREAD_MUTEX_LOCK(&B->mutex);
    while(B->count == BUFF_SIZE)
        PTHREAD_COND_WAIT(&B->buffer_not_full, &B->mutex);
    /* put data in the buffer and update count and last */
    PTHREAD_COND_SIGNAL(&B->buffer_not_empty);
    PTHREAD_MUTEX_UNLOCK(&B->mutex);
    return 0;
}

int take(int *item, buffer *B ) {
    PTHREAD_MUTEX_LOCK(&B->mutex);
    while(B->count == 0)
        PTHREAD_COND_WAIT(&B->buffer_not_empty, &B->mutex);
    /* get data from the buffer and update count and first */
    PTHREAD_COND_SIGNAL(&B->buffer_not_full);
    PTHREAD_MUTEX_UNLOCK(&B->mutex);
    return 0;
}

/* an initialize() function is also required */
```

Although mutexes and condition variables act as a type of monitor, their semantics do differ when a thread is released from a conditional wait and other threads are trying

to gain access to the critical region. With C/Real-Time POSIX, it is unspecified which thread succeeds unless priority-based scheduling is being used (see Section 12.6).

Read/write locks and barriers

Mutexes are mutual exclusion locks that allow threads read and write access to the shared data. On occasions more flexible locking is required. For example, a thread may only require a lock to read the data. Hence, multiple threads that only wish to read the data can access it concurrently. For these occasions, C/Real-Time POSIX provides **read/write locks**. They are similar to mutexes except:

- the lock operation specifies whether a read or a write lock is required (`pthread_rwlock_rdlock`, `pthread_rwlock_wrlock`);
- the full range of attributes are not supported – for example there is no support for priority inversion avoidance (see Section 11.8).

The other useful mechanism that supports C/Real-Time POSIX pthreads are **barriers**. A barrier is a simple mechanism that allows threads to be blocked until a number of them have arrived at the barrier. As with all pthread mechanisms they have attributes and an initialize function. The barrier is initialized with the number of threads required. The threads then call the `pthread_barrier_wait` function and the function does not return until the required number have arrived.

5.8 Protected objects in Ada

The criticism of monitors centres on their use of condition variables. By replacing this approach to synchronization by the use of guards, a more structured abstraction is obtained. This form of monitor will be termed a **protected object**. Ada is the only major language that provides this mechanism, and hence it will be described in terms of Ada.

A protected object in Ada encapsulates data items and allows access to them only via protected subprograms or protected entries. The language guarantees that these subprograms and entries will be executed in a manner that ensures that the data is updated under mutual exclusion. Condition synchronization is provided by having boolean expressions on entries (these are guards but are termed **barriers** in Ada) that must evaluate to True before a task is allowed entry. Consequently, protected objects are rather like monitors and conditional critical regions. They provide the structuring facility of monitors with the high-level synchronization mechanism of conditional critical regions.

A protected unit may be declared as a type or as a single instance; it has a specification and a body (hence it is declared in a similar way to a task). Its specification may contain functions, procedures and entries.

The following declaration illustrates how protected types can be used to provide simple mutual exclusion:

```
-- a simple integer
protected type Shared_Integer(Initial_Value : Integer) is
  function Read return Integer;
  procedure Write(New_Value : Integer);
```



```

procedure Increment (By : Integer);
private
  The_Data : Integer := Initial_Value;
end Shared_Integer;

My_Data : Shared_Integer(42);

```

The above protected type encapsulates a shared integer. The object declaration `My_Data` declares an instance of the protected type and passes the initial value for the encapsulated data. The encapsulated data can now only be accessed by the three subprograms: `Read`, `Write` and `Increment`.

A protected procedure provides mutually exclusive *read/write* access to the data encapsulated. In this case, concurrent calls to the procedure `Write` or `Increment` will be executed in mutual exclusion; that is, only one can be executing at any one time.

Protected functions provide concurrent *read-only* access to the encapsulated data. In the above example, this means that many calls to `Read` can be executed simultaneously. However, calls to a protected function are still executed mutually exclusively with calls to a protected procedure. A `Read` call cannot be executed if there is a currently executing procedure call; a procedure call cannot be executed if there are one or more concurrently executing function calls. The body of the `Shared_Integer` is simply:

```

protected body Shared_Integer is
  function Read return Integer is
    begin
      return The_Data;
    end Read;

  procedure Write(New_Value : Integer) is
    begin
      The_Data := New_Value;
    end Write;

  procedure Increment (By : Integer) is
    begin
      The_Data := The_Data + By;
    end Increment;
end Shared_Integer;

```

A protected entry is similar to a protected procedure in that it is guaranteed to execute in mutual exclusion and has *read/write* access to the encapsulated data. However, a protected entry is guarded by a boolean expression (the barrier) inside the body of the protected object; if this barrier evaluates to `False` when the entry call is made, the calling task is suspended until the barrier evaluates to `True` and no other tasks are currently active inside the protected object. Hence protected entry calls can be used to implement condition synchronization.

Consider a bounded buffer shared between several tasks. The specification of the buffer is:

```

-- a bounded buffer

Buffer_Size : constant Integer := 10;

```



```

type Index is mod Buffer_Size ;
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer is array (Index) of Data_Item;

protected type Bounded_Buffer is
  entry Get(Item: out Data_Item);
  entry Put(Item: in Data_Item);
private
  First : Index := Index'First;
  Last : Index := Index'Last;
  Number_In_Buffer : Count := 0;
  Buf : Buffer;
end Bounded_Buffer;

My_Buffer : Bounded_Buffer;

```

Two entries have been declared; these represent the public interface of the buffer. The data items declared in the private part are those items which must be accessed under mutual exclusion. In this case, the buffer is an array and is accessed via two indices; there is also a count indicating the number of items in the buffer. The body of this protected type is given below.

```

protected body Bounded_Buffer is
  entry Get(Item: out Data_Item)
    when Number_In_Buffer /= 0 is
  begin
    Item := Buf(First);
    First := First + 1; -- mod types cycle around
    Number_In_Buffer := Number_In_Buffer - 1;
  end Get;

  entry Put(Item: in Data_Item)
    when Number_In_Buffer /= Buffer_Size is
  begin
    Last := Last + 1;
    Buf(Last) := Item;
    Number_In_Buffer := Number_In_Buffer + 1;
  end Put;
end Bounded_Buffer;

```

The Get entry is guarded by the barrier '**when** Number_In_Buffer /= 0'; only when this evaluates to True can a task execute the Get entry; similarly with the Put entry. Barriers define a precondition; only when they evaluate to True can the entry be accepted.

Although calls to a protected object can be delayed because the object is in use (that is, they cannot be executed with the requested read or read/write access), Ada does not view the call as being suspended. Calls which are delayed due to an entry barrier being false are, however, considered suspended and placed on a queue. The reason for this is:

- it is assumed that protected operations are short-lived;

- once started a protected operation cannot suspend its execution – all calls which are potentially suspending are prohibitive and raise exceptions – it can only requeue (see Section 8.4).

Hence a task should not be delayed for a significant period while attempting to access the protected object – other than for reasons associated with the order of scheduling. Once a procedure (or function) call has gained access it will immediately start to execute the subprogram; an entry call will evaluate the barrier and will, of course, be blocked if the barrier is false. In Section 12.3, the implementation strategy required by the Real-Time Systems Annex is considered which guarantees that a task is never delayed when trying to gain access to a protected object.

5.8.1 Entry calls and barriers

To issue a call to a protected object, a task simply names the object and the required subprogram or entry. For example, to place some data into the above bounded buffer requires the calling task to:

```
My_Buffer.Put(Some_Item);
```

At any instant in time, a protected entry is either open or closed. It is open if, when checked, the boolean expression evaluates to `True`; otherwise it is closed. Generally, the protected entry barriers of a protected object are evaluated when:

- (1) a task calls one of its protected entries and the associated barrier references a variable or an attribute which might have changed since the barrier was last evaluated;
- (2) a task leaves a protected procedure or protected entry and there are tasks queued on entries whose barriers reference variables or attributes which might have changed since the barriers were last evaluated.

Barriers are not evaluated as a result of a protected function call. Note that it is not possible for two tasks to be active within a protected entry or procedure as the barriers are only evaluated when a task leaves the object.

When a task calls a protected entry or a protected subprogram, the protected object may already be locked: if one or more tasks are executing protected functions inside the protected object, the object is said to have an active **read lock**; if a task is executing a protected procedure or a protected entry, the object is said to have an active **read/write lock**.

If more than one task calls the same closed barrier then the calls are queued, by default, in a first-come, first-served fashion. However, this default can be changed (see Section 12.3).

Two more examples will now be given. Consider first the simple resource controller given earlier. When only a single resource is requested (and released) the code is straightforward:

```
protected Resource_Control is
  entry Allocate;
  procedure Deallocate;
```



```

private
  Free : Boolean := True;
end Resource_Control;

protected body Resource_Control is
  entry Allocate when Free is
  begin
    Free := False;
  end Allocate;
  procedure Deallocate is
  begin
    Free := True;
  end Deallocate;
end Resource_Control;

```

The resource is initially available and hence the `Free` flag is true. A call to `Allocate` changes the flag, and therefore closes the barrier; all subsequent calls to `Allocate` will be blocked. When `Deallocate` is called, the barrier is opened. This will allow one of the waiting tasks to proceed by executing the body of `Allocate`. The effect of this execution is to close the barrier again, and hence no further executions of the entry body will be possible (until there is a further call of `Deallocate`).

Interestingly, the general resource controller (where groups of resources are requested and released) is not easy to program using just guards. The reasons for this will be explained in Chapter 8, where resource control is considered in some detail.

Each entry queue has an attribute associated with it that indicates how many tasks are currently queued. This is used in the following example. Assume that a task wishes to broadcast a value (of type `Message`) to a number of waiting tasks. The waiting tasks will call a `Receive` entry which is only open when a new message has arrived. At that time, all waiting tasks are released.

Although all tasks can now proceed, they must pass through the protected object in strict sequence (as only one can ever be active in the object). The last task out must then set the barrier to false again so that subsequent calls to `Receive` are blocked until a new message is broadcast. This explicit setting of the barriers can be compared with the use of condition variables which have no lasting effect (within the monitor) once all tasks have exited. The code for the broadcast example is as follows (note that the attribute `Count` indicates the number of tasks queued on an entry):

```

protected type Broadcast is
  entry Receive(M : out Message);
  procedure Send(M : Message);
private
  New_Message : Message;
  Message_Arrived : Boolean := False;
end Broadcast;

protected body Broadcast is

  entry Receive(M : out Message) when Message_Arrived is
  begin
    M := New_Message;
    if Receive'Count = 0 then

```



```

    Message_Arrived := False;
  end if;
end Receive;

procedure Send(M : Message) is
begin
  if Receive'Count > 0 then
    Message_Arrived := True;
    New_Message := M;
  end if;
end Send;

end Broadcast;
```

As there may be no tasks waiting for the message, the send procedure has to check the Count attribute. Only if it is greater than zero will it set the barrier to true (and record the new message).

Finally, this section gives a full Ada implementation of the semaphore package given in Section 5.4.5. This shows that protected objects are not only an excellent structuring abstraction but have the same expressive power as semaphores.

```

package Semaphore_Package is
  type Semaphore(Initial : Natural := 1) is limited private;
  procedure Wait (S : in out Semaphore);
  procedure Signal (S : in out Semaphore);
private
  protected type Semaphore(Initial : Natural := 1) is
    entry Wait_Imp;
    procedure Signal_Imp;
  private
    Value : Natural := Initial;
  end Semaphore;
end Semaphore_Package;

package body Semaphore_Package is
  protected body Semaphore is
    entry Wait_Imp when Value > 0 is
    begin
      Value := Value - 1;
    end Wait_Imp;

    procedure Signal_Imp is
    begin
      Value := Value + 1;
    end Signal_Imp;
  end Semaphore;

  procedure Wait(S : in out Semaphore) is
  begin
    S.Wait_Imp;
  end Wait;

  procedure Signal(S : in out Semaphore) is
  begin
```



```

    S.Signal_Imp;
  end Signal;
end Semaphore_Package;

```

5.8.2 Protected objects and object-oriented programming

As mentioned in Section 4.4.4 Ada 95 did not attempt to integrate the language's support for concurrent programming directly into the OOP model. Instead, the models were orthogonal and paradigms had to be created to allow the benefits of OOP to be available in a concurrent environment. These paradigms had inherent limitations and proposals were developed to add OOP facilities directly into, for example, the protected type mechanism. Unfortunately, these proposals added another layer of complexity to the language and they did not receive widespread support. The introduction of interfaces into Ada 2005 allows the concurrency facilities to provide some limited support of inheritance.

Interfaces in Ada 2005 are classified according to the type of object that can be supported ('implemented' using the Java terminology). For the purpose of this book the following interfaces are relevant.

- **Synchronized** – this specifies a collection of functions and procedures that can be implemented by a task type or a protected type.
- **Protected** – this specifies a collection of functions and procedures that can only be implemented by a protected type.
- **Task** – this specifies a collection of functions and procedures that can only be implemented by a task type.

Synchronized and protected interfaces will be considered in this section; discussion of task interfaces is deferred until Section 6.3.3.

The key idea of a synchronized interface is that there is some implied synchronization between the task that calls an operation from an interface and the object that implements the interface. Synchronization in Ada is achieved via two main mechanisms: a protected action (call of an entry or protected subprogram – a shared variable-based communication mechanism) or the rendezvous (a message-based communication mechanism – see Chapter 6). Hence, a task type or a protected type can implement a synchronized interface. Both protected entries and procedures can implement a synchronized interface procedure. A protected function can implement a synchronized interface function.

Where the programmer is not concerned with the form of synchronization, a synchronized interface is the appropriate abstraction. For situations where the programmer requires a particular form of synchronization, protected interfaces or task interfaces should be used explicitly. For example, there are various communication paradigms that all have at their heart some form of buffer. They, therefore, all have buffer-like operations in common. Some programs will use these paradigms and will not care whether the implementation uses a mailbox, a link or whatever. Some will require a task in the implementations, others will just need a protected object. Synchronized interfaces allow the programmer to defer the commitment to a particular paradigm and its implementation approach.

Consider the operations that can be performed on all integer buffers:

```
package Integer_Buffers is
  type Buffer is synchronized interface;
  procedure Put (Buf : in out Buffer; Item : in Integer)
    is abstract;
  procedure Get (Buf : in out Buffer; Item : out Integer)
    is abstract;
end Integer_Buffers;
```

In the above code, the Buffer type declaration indicates that it is a synchronized interface, which supports the Put and Get procedures.

Now consider a protected type that can implement this interface.

```
with Integer_Buffers;
package Integer_Buffers.MailBoxes is
  subtype Capacity_Range is range ...;
  subtype Count is Integer range ...;
  type Buffer_Store is array (Capacity_Range) of Integer;

  protected type Mailbox is new Buffer with
    overriding entry Put (Item : in Integer);
    overriding entry Get (Item : out Integer);
  private
    First : Capacity_Range := Capacity_Range'first;
    Last : Capacity_Range := Capacity_Range'last;
    Number_In_Buffer : Count := 0;
    Box_Store : Buffer_Store;
  end Mailbox;
end Integer_Buffers.Mailboxes;
```

Here, the declaration of the Mailbox protected type indicates that it implements the Buffer interface by being derived from that type. The 'overriding' keyword on the entries indicates that these entries implement the interface's procedures. The name of the entry and its parameter type must match the interface's. Note, however, because of the way OOP is supported in Ada, the Buf parameter is not required as its type is the same as the type that the Mailbox is derived from. Hence, it is an implicit parameter that is generated when an instance of the mailbox is used. For example, in

```
with Integer_Buffers.Mailboxes;
use Integer_Buffers.Mailboxes;
...
Mail : Mailbox;
...
Mail.Put(42);
```

the Mail object provides the implicit first parameter.

The main limitations with the Ada approach is that you cannot derive one protected type from another.

5.9 Synchronized methods in Java

In many ways, Ada's protected objects are like objects in a class-based object-oriented programming language. The main difference, of course, is that they do not support a full inheritance relationship. Java, having a fully integrated concurrency and object-oriented model, provides a mechanism by which monitors can be implemented in the context of classes and objects.

In Java, there is a lock associated with each object. This lock cannot be accessed directly by the application, but it is affected by:

- the method modifier `synchronized`; and
- block synchronization.

When a method is labelled with the `synchronized` modifier, access to the method can only proceed once the lock associated with the object has been obtained. Hence synchronized methods have mutually exclusive access to the data encapsulated by the object, *if that data is only accessed by other synchronized methods*. Non-synchronized methods do not require the lock, and can therefore be called at any time. Hence to obtain full mutual exclusion, every method has to be labelled `synchronized`. A simple shared integer is therefore represented by:

```
class SharedInteger
{
    private int theData;

    public SharedInteger(int initialValue) {
        theData = initialValue;
    }

    public synchronized int read() {
        return theData;
    }

    public synchronized void write(int newValue) {
        theData = newValue;
    }

    public synchronized void incrementBy(int by) {
        theData = theData + by;
    }
}
```

```
SharedInteger myData = new SharedInteger(42);
```

Block synchronization provides a mechanism whereby a block can be labelled as `synchronized`. The `synchronized` keyword takes as a parameter an object whose lock it needs to obtain before it can continue. Hence synchronized methods are effectively implementable as:

```
public int read() {
    synchronized(this) {
```



```

    return theData;
}
}

```

where **this** is the Java mechanism for obtaining the current object.

Used in its full generality, the synchronized block can undermine one of the advantages of monitor-like mechanisms: that of encapsulating synchronization constraints associated with an object into a single place in the program. This is because it is not possible to understand the synchronization associated with a particular object by just looking at the object itself when other objects can name that object in a synchronized statement. However, with careful use this facility augments the basic model and allows more expressive synchronization constraints to be programmed, as will be shown shortly.

Although synchronized methods or blocks allow mutually exclusive access to data in an object, this is not adequate if that data is *static*. Static data is shared between all objects created from the class. To obtain mutually exclusive access to this data requires access to a different lock.

In Java, classes themselves are also objects and therefore there is a lock associated with the class. This lock may be accessed either by labelling a static method with the synchronized modifier or by identifying the class's object in a synchronized block statement. The latter can be obtained from the `Object` class associated with the object. Note, however, that this class-wide lock is not obtained when synchronizing on the object. Hence to obtain mutual exclusion over a static variable requires the following (for example):

```

class StaticSharedVariable
{
    private static int shared;
    ...

    public synchronized static int Read() {
        return shared;
    }

    public synchronized static void Write(int I) {
        shared = I;
    }
}

```

5.9.1 Waiting and notifying

To obtain conditional synchronization requires further support. This again comes from methods provided in the predefined `Object` class as illustrated in Program 5.5. These methods are designed to be used only from within methods that hold the object lock (i.e. they are synchronized). If called without the lock, the exception `IllegalMonitorStateException` is thrown.

The `wait` method always blocks the calling thread and releases the lock associated with the object. If the call is made from within a nested monitor then only the lock associated with the `wait` is released.

Program 5.5 Support for waiting and notifying in the `Object` class.

```

package java.lang;
public class Object {
    ...
    // The following methods all throw the unchecked
    // IllegalMonitorStateException.
    public final void notify();
    public final void notifyAll();
    public final void wait() throws InterruptedException;
}

```

The `notify` method wakes up one waiting thread; the one woken is not defined by the Java language (however, it is defined by Real-Time Java; see Section 12.7). Note that `notify` does not release the lock, and hence the woken thread must still wait until it can obtain the lock before it can continue. To wake up *all* waiting threads requires use of the `notifyAll` method; again this does not release the lock and all the awoken threads must contend for the lock when it becomes free. If no thread is waiting, then `notify` and `notifyAll` have no effect.

A waiting thread can also be awoken if it is interrupted by another thread. In this case the `InterruptedException` is thrown. This situation will be ignored in this chapter (the exception will be allowed to propagate), but discussed fully in Section 7.7.2.

Although it appears that Java provides the equivalent facilities to other languages supporting monitors, there is one important difference. There are no explicit condition variables. Hence, when a thread is awoken, it cannot necessarily assume that its ‘condition’ is true, as all threads are potentially awoken irrespective of what conditions they were waiting on. For many algorithms this limitation is not a problem, as the conditions under which tasks are waiting are mutually exclusive.

For example, the bounded buffer traditionally has two condition variables: `BufferNotFull` and `BufferNotEmpty`, each associated with the corresponding buffer state. If a thread is waiting for one condition, no other thread can be waiting for the other condition as the buffer cannot be both full and empty at the same time. Hence, one would expect that the thread can assume that when it wakes, the buffer is in the appropriate state. Unfortunately, this is not always the case. Java, in common with other monitor-like approaches (for example, C/Real-Time POSIX mutexes), makes no guarantee that a thread woken from a wait will gain immediate access to the lock. Furthermore, a Java implementation is allowed to generate spurious wake-ups not related to the application.

Consider a thread that is woken after waiting on the `BufferNotFull` condition. Another thread could call the `put` method, find that the buffer has space and insert data into the buffer. When the woken thread eventually gains access to the lock, the buffer will again be full. Hence, it is usually essential for threads to re-evaluate their conditions, as illustrated in the integer bounded buffer example below.

```

public class BoundedBuffer {
    public BoundedBuffer(int length) {

```



```

    size = length;
    buffer = new int[size];
    last = 0;
    first = 0;
}
public synchronized void put(int item)
    throws InterruptedException {
    while (numberInBuffer == size) wait();
    last = (last + 1) % size;
    numberInBuffer++;
    buffer[last] = item;
    notifyAll();
}
public synchronized int get()
    throws InterruptedException {
    while (numberInBuffer == 0) wait();
    first = (first + 1) % size;
    numberInBuffer--;
    notifyAll();
    return buffer[first];
}
private int buffer[];
private int first;
private int last;
private int numberInBuffer = 0;
private int size;
}

```

Of course, if `notifyAll` is used to wake up threads, then it is more obvious that those threads must always re-evaluate their conditions before proceeding.

In general, many simple synchronization errors can be avoided in Java if all `wait` method calls are enclosed in while loops that evaluate the waiting conditions and the `notifyAll` method is used to signal changes to each object's state. This approach, while safe, is potentially inefficient as spurious wake-ups will occur. To improve performance, the `notify` method may be used when:

- all threads are waiting for the same condition;
- at most one waiting thread can benefit from the state change;
- the JVM does not generate any wake-ups without an associated call to the `notify` and `notifyAll` methods on the corresponding object.

The readers–writers problem

One of the standard concurrency control problems is the **readers–writers** problem. In this, many readers and many writers are attempting to access a large data structure. Readers can read concurrently, as they do not alter the data; however, writers require mutual exclusion over the data both from other writers and from readers. There are different variations on this scheme; the one considered here is where priority is always given to waiting writers. Hence, as soon as a writer is available, all new readers will be blocked until all writers have finished. Of course, in extreme situations this may lead to starvation of readers.

The solution to the readers–writers problem using standard monitors requires four monitor procedures – `startRead`, `stopRead`, `startWrite` and `stopWrite`. The readers are structured:

```
startRead();
    // read data structure
stopRead();
```

Similarly, the writers are structured:

```
startWrite();
    // write data structure
stopWrite();
```

The code inside the monitor provides the necessary synchronization using two condition variables: `OkToRead` and `OkToWrite`. In Java, the approach is as follows.

```
public class ReadersWriters {
    // Preference is given to waiting writers.
    public synchronized void startWrite()
        throws InterruptedException {
        // Wait until it is ok to write.
        while(readers > 0 || writing) {
            waitingWriters++;
            try {
                wait();
            } finally { waitingWriters--; }
        }
        writing = true;
    }
    public synchronized void stopWrite() {
        writing = false;
        notifyAll();
    }
    public synchronized void startRead()
        throws InterruptedException {
        // Wait until it is ok to read.
        while(writing || waitingWriters > 0) wait();
        readers++;
    }
    public synchronized void stopRead() {
        readers--;
        if(readers == 0) notifyAll();
    }
    private int readers = 0;
    private int waitingWriters = 0;
    private boolean writing = false;
}
```

In this solution, on awaking after the wait request, the thread must re-evaluate the conditions under which it can proceed. Although this approach will allow multiple readers or a single writer, arguably it is inefficient, as all threads are woken up every time the data becomes available. Many of these threads, when they finally gain access to the monitor, will find that they still cannot continue and therefore will have to wait again.

Program 5.6 An abridged Java Lock interface.

```

package java.util.concurrent.locks;
public interface Lock {
    public void lock();
        // Uninterruptibly wait for the lock to be acquired.
    public void lockInterruptibly()
        throws InterruptedException;
        // As above but interruptible.
    public Condition newCondition();
        // Create a new condition variable for use with the Lock.
    public boolean tryLock();
        // Returns true is lock is available immediately.
    public void unlock();
    ...
}

```

Program 5.7 An abridged Java Condition interface.

```

package java.util.concurrent.locks;
public interface Condition {
    public void await() throws InterruptedException;
    /* Atomically releases the associated lock and
     * causes the current thread to wait until
     * 1. another thread invokes the signal method
     *    and the current thread happens to be chosen
     *    as the thread to be awakened; or
     * 2. another thread invokes the signalAll method;
     * 3. another thread interrupts the thread; or
     * 4. a spurious wake-up occurs.
     * When the method returns it is guaranteed to hold the
     * associated lock.
     */

    public void awaitUninterruptible();
        // As for await, but not interruptible.
    public void signal();
        // Wake up one waiting thread.
    public void signalAll();
        // Wake up all waiting threads.
}

```

5.9.2 Synchronizers and locks

The Java concurrency utilities provide a range of support packages aimed at easing the burden of concurrent programming. These expand the built-in facilities that have been discussed above. The package that is most relevant to this section is `java.util.concurrent.locks`. Its main goal is to provide efficient support for

Program 5.8 An abridged Java ReentrantLock class.

```

package java.util.concurrent.locks;

public class ReentrantLock implements Lock, java.io.Serializable {
    public ReentrantLock();

    ...
    public void lock();
    public void lockInterruptibly() throws InterruptedException;
    public Condition newCondition();
        // Create a new condition variable and associated it
        // with this lock object.
    public boolean tryLock();
    public void unlock();
}

```

various locking approaches including locks that support explicit condition variables. Programs 5.6 and 5.7 show the Java interfaces that support these abstractions. Various types of lock are provided, including mutual exclusion locks (the `ReentrantLock` class shown in Program 5.8) and read/write locks.

Using these facilities it is possible to implement the bounded buffer using the familiar algorithm with two condition variables.

```

import java.util.concurrent.locks.*;
public class BoundedBuffer2 {
    public BoundedBuffer2(int length) {
        size = length;
        buffer = new int[size];
        last = 0;
        first = 0;
        numberInBuffer = 0;
        lock = new ReentrantLock();
        notFull = lock.newCondition();
        notEmpty = lock.newCondition();
    }
    public void put(int item)
        throws InterruptedException {
        lock.lock();
        try {
            while (numberInBuffer == size) notFull.await();
            last = (last + 1) % size;
            numberInBuffer++;
            buffer[last] = item;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }
}

```

```

public synchronized int get()
    throws InterruptedException {
    lock.lock();
    try {
        while (numberInBuffer == 0) notEmpty.await();
        first = (first + 1) % size ;
        numberInBuffer--;
        notFull.signal();
        return buffer[first];
    } finally {
        lock.unlock();
    }
}

private int buffer[];
private int first;
private int last;
private int numberInBuffer;
private int size;
private Lock lock;
private final Condition notFull;
private final Condition notEmpty;
}

```

Note the use of a finally clause to ensure the unlock method is always called before the method exits.

5.9.3 Inheritance and synchronization

The combination of the object-oriented paradigm with mechanisms for concurrent programming may give rise to the so-called **inheritance anomaly** (Matsuoka and Yonezawa, 1993). An inheritance anomaly exists if the synchronization between operations of a class is not local but may depend on the whole set of operations present for the class. When a subclass adds new operations, it may become necessary to change the synchronization defined in the parent class to account for these new operations.

For example, consider the bounded buffer presented earlier in this section. With Java, the code that tests the conditions (`BufferNotFull` and `BufferNotEmpty`) is embedded in the methods. It will be shown in Chapter 8 that, in general, this approach has many advantages as it allows the methods to access the parameters to the method as well as the object attributes. However, it does cause some problems when inheritance is considered. Suppose that the bounded buffer is to be subclassed so that all accesses can be prohibited. Two new methods are added, `prohibitAccess` and `allowAccess`. A naive extension might include the following code:

```

public class LockableBoundedBuffer extends BoundedBuffer {
    boolean prohibited ;

    // Incorrect Code

    LockableBoundedBuffer(int length) {
        super(length);
        prohibited = false;
    }
}

```



```

public synchronized void prohibitAccess() throws InterruptedException {
    while (prohibited) wait();
    prohibited = true;
}

public synchronized void allowAccess() throws AccessError {
    if (!prohibited) throw new AccessError();
    prohibited = false;
    notifyAll();
}

public synchronized void put(int item) throws InterruptedException {
    while (prohibited) wait();
    super.put(item);
}

public synchronized int get() throws InterruptedException {
    while (prohibited) wait();
    return (super.get());
}
}

```

Unfortunately, there is a subtle bug with this approach. Consider the case where a producer is attempting to put data into a full buffer. Access to the buffer is not prohibited so `super.put(item)` is invoked where the call is blocked waiting for the `BufferNotEmpty` condition. Now access to the buffer is prohibited. Any further calls to `get` and `put` are held in the overridden subclass methods, as will be further calls to the `prohibitAccess` method. Now a call to the `allowAccess` method is made; this results in all waiting threads being released. Suppose the order in which the released threads acquire the monitor lock is: the consumer thread, the thread attempting to prohibit access to the buffer, the producer thread. The consumer finds that access to the buffer is not prohibited and takes data from the buffer. It issues a `notifyAll` request, but no threads are now currently waiting. The next thread which runs now prohibits access to the buffer. The producer thread runs next and places an item into the buffer although access is prohibited!

Although this example might seem contrived, it does illustrate the subtle bugs that can occur due to the inheritance anomaly.

5.10 Shared memory multiprocessors

Multiprocessor systems are becoming more prevalent. In particular symmetric multiprocessor (SMP) systems, where processors have shared access to the main memory, are often the default platform for large real-time systems rather than a single processor system. From a theoretical concurrent programming viewpoint, a program that is properly synchronized and executes successfully on a single processor systems will execute successfully on a SMP system. Programs that are not properly synchronized may suffer from data race conditions (see Section 5.3). Even properly synchronized programs can suffer from deadlocks. Consequently, a program that *appears* to execute correctly on a single processor cannot be guaranteed to work correctly on a multiprocessor as it is doubtful that all possible interleaving of task executions will have been exercised on

the single processor system. As a result, concurrency-related faults/bugs will remain dormant.

In order to understand how a concurrent program executes on a SMP system it is necessary to understand the memory consistency model provided by the machine. The simplest model is *sequential consistency*. An SMP system is sequentially consistent if the result of any execution of a program is the same as if the instructions of all the processors are executed in some sequential order, and the instructions of any thread within the sequence is the same as that specified by its program logic (Lamport, 1997). Hence, both atomicity of instructions and the maintenance of task instruction sequences are required.

Sequential consistency is a very restrictive property, and if rigidly supported would disallow many optimizations that are typically performed by compilers and modern multiprocessors. For example, a compiler would not be able to reorder the instructions and the hardware would not be able to execute instructions out of order. The presence of hardware caches exacerbates these problems.

To overcome the severe constraints imposed by the requirement of sequential consistency, **relaxed memory models** can be used. These either relax the instruction order or the atomicity requirements. Different SMP architectures adopt different approaches – see Adve and Gharachorloo (1996) for a classification.

From the programmer's perspective, it is crucial to understand what guarantees a programming language provides when a shared variable is updated, in particular, when that update becomes visible to other tasks potentially executing on other processors. The remainder of this section considers the guarantees provided by Java and Ada. It is the compiler and the run-time systems that must implement these guarantees on the underlying architecture's memory model irrespective of the memory model it provides.

5.10.1 The Java memory model

Early versions of the Java language were criticized because its semantics on multiprocessors were ill defined and had serious problems (Pugh, 2000). The Java 5 language has corrected this with a new memory model (the Java Memory Model, JMM) that, on the one hand allows both compiler and hardware optimization, but, on the other hand, gives intuitive semantics to program code.

From a language semantics view point, a concurrent program can be defined using a trace model. A trace model defines the meaning of a thread as the set of sequences of events (traces) that the thread can be observed to perform. Hence, a program's execution is the set of all possible thread traces. The language's **memory model** describes, given a program and an execution trace of that program, whether the execution trace is a legal execution of the program.

The JMM is concerned with **memory actions**, which are defined to be reads and writes to memory locations shared between threads. Local variables, formal method parameters and exception handler parameters are never shared, and consequently fall outside the model. Given two actions *A* and *B*, the results of action *A* is visible to action *B* if there is a **happens-before** relation between them. The following defines the relation:

- if *A* and *B* are in the same thread and *A* comes before *B* in the program order then *A* happens-before *B*;

- an unlock monitor action happens-before all subsequent lock actions on the same monitor;
- a write to a volatile¹ variable V happens-before all subsequent reads from V in any thread;
- an action that starts a thread happens-before the first action of the thread it starts;
- the final action of a thread happens-before any action in any other thread that determines that it has terminated (using `isAlive` or the `join` methods in the `Thread` class);
- the interruption of a thread T (via the `interrupt` method in the `Thread` class) happens-before any thread that detects that T has been interrupted (via the `interrupted` and `isInterrupt` methods in the `Thread` class or by having the `InterruptedException` thrown);
- if A happens-before B and B happens-before C , then A happens-before C .

The formal semantics of the JMM are complex; however, they can be approximated by the following two rules.

- The actions of each thread in isolation are defined by the action of its code (in program order) in isolation, with the exception that the values seen by each read variable action are determined by the JMM.
- A read operation on variable A must return the value written to it by the previous write operation that *happened before* it.

From this, the following points need to be emphasized when accessing variables shared between threads.

- When one thread starts another – changes made by the parent thread before the start requests are visible to the child thread when it executes.
- When one thread waits for the termination of another – changes made by the terminating thread before it terminates are visible to the waiting thread once termination has been detected.
- When one thread interrupts another – changes made by the interrupting thread before the interrupt request are made visible to the interrupted thread when the interruption is detected by the interrupted thread.
- When threads read and write to the same volatile field – changes made by the writer thread to shared data (before it writes to the volatile field) are made visible to a subsequent reader of the same volatile field.

Any implementation of Java on a SMP system must respect the Java Memory Model, and where the architecture potentially performs optimization that might undermine it, code must be executed to ensure that this does not occur (for example memory fences or barriers must be inserted).

¹A **volatile** variable is one that cannot be held in local registers or caches. All read and write operations go directly to the memory.

5.10.2 Ada and shared variables

Ada has no explicit memory model but, like Java, the Ada language defines the conditions under which it is safe to read and write to shared variables outside the rendezvous or protected objects. Hence, the model is implicit.

The safe conditions are as follows:

- where one task writes a variable before activating another task that reads the variable;
- where the activation of one task writes the variable and the task awaiting completion of the activation reads the variable;
- where one task writes the variable and another task waits for the termination of the task and then reads the variable;
- where one task writes the variable before making an entry call on another task, and the other task reads the variable during the corresponding entry body or accept statement;
- where one task writes a shared variable during an accept statement and the calling task reads the variable after the corresponding entry call has returned;
- where one task writes a variable whilst executing a protected procedure body or entry, and the other task reads the variable, for example as part of a later execution of an entry body of the same protected body.

If the Systems Programming Annex is supported, there are extra facilities that can be used to control shared variables between *unsynchronized* tasks. They come in the form of extra pragmas which can be applied to certain objects or type declarations.

- `Pragma Volatile` – `pragma Volatile` ensures that all reads and writes go directly to memory.
- `Pragma Volatile_Components` – `pragma Volatile_Components` applies to components of an array.
- `Pragma Atomic` and `pragma Atomic_Components` – whilst `pragma Volatile` indicates that all reads and writes must be directed straight to memory, `pragma Atomic` imposes the further restriction that they must be indivisible. That is, if two tasks attempt to read and write the shared variable at the same time, then the result must be internally consistent. An implementation is not required to support atomic operations for all types of variable; however, if not supported for a particular object, the pragma (and hence the program) must be rejected by the compiler.

The language defines accesses to volatile and atomic variables to be interactions with the external environment, and hence compilers must ensure that no reordering of instructions occurs across their use.

Unlike Java, which attempts to define the semantics of a program that is not properly synchronized, Ada simply defines these situations to result in erroneous program execution.

5.11 Simple embedded system revisited

In Section 4.8, a simple embedded system was introduced and a concurrent solution was proposed. The Ada solution is now updated to illustrate communication with the operator console. Recall that the structure of the controller is as below:

```
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;

procedure Controller is
  task Temp_Controller;
  task Pressure_Controller;

  task body Temp_Controller is
    TR : Temp_Reading; HS : Heater_Setting;
  begin
    loop
      Read(TR);
      Temp_Convert(TR, HS);
      Write(HS);
      Write(TR);
    end loop;
  end Temp_Controller;

  task body Pressure_Controller is
    PR : Pressure_Reading; PS : Pressure_Setting;
  begin
    loop
      Read(PR);
      Pressure_Convert(PR, PS);
      Write(PS);
      Write(PR);
    end loop;
  end Pressure_Controller;

begin
  null;    -- Temp_Controller and Pressure_Controller
          -- have started their executions
end Controller;
```

and that the interfaces to the I/O routines were:

```
with Data_Types; use Data_Types;
package IO is
  -- procedures for data exchange with the environment
  procedure Read(TR : out Temp_Reading); -- from DAC
  procedure Read(PR : out Pressure_Reading); -- from DAC
  procedure Write(HS : Heater_Setting); -- to switch.
  procedure Write(PS : Pressure_Setting); -- to DAC
  procedure Write(TR : Temp_Reading); -- to console
  procedure Write(PR : Pressure_Reading); -- to console
end IO;
```

The body of the I/O routines can now be completed. The data to be sent to the console is stored in a monitor (in this case using an Ada protected object). The console task will call the entry to get the new data.

```

package body IO is
  task Console;
  protected Console_Data is
    procedure Write(R : Temp_Reading);
    procedure Write(R : Pressure_Reading);
    entry Read(TR : out Temp_Reading;
              PR : out Pressure_Reading);
  private
    Last_Temperature : Temp_Reading;
    Last_Pressure : Pressure_Reading;
    New_Reading : Boolean := False;
  end Console_Data;

  -- procedures for data exchange with the environment
  procedure Read(TR : out Temp_Reading) is separate; -- from DAC
  procedure Read(PR : out Pressure_Reading) is separate; -- from DAC
  procedure Write(HS : Heater_Setting) is separate; -- to switch.
  procedure Write(PS : Pressure_Setting) is separate; -- to DAC

  task body Console is
    TR : Temp_Reading;
    PR : Pressure_Reading;
  begin
    loop
      ...
      Console_Data.Read(TR, PR);
      -- Display new readings
    end loop;
  end Console;

  protected body Console_Data is
    procedure Write(R : Temp_Reading) is
    begin
      Last_Temperature := R;
      New_Reading := True;
    end Write;

    procedure Write(R : Pressure_Reading) is
    begin
      Last_Pressure := R;
      New_Reading := True;
    end Write;

    entry Read(TR : out Temp_Reading;
              PR : out Pressure_Reading)
      when New_Reading is
    begin
      TR := Last_Temperature;
      PR := Last_Pressure;
      New_Reading := False;
    end Read;
  end Console_Data;

```



```

procedure Write(TR : Temp_Reading) is
begin
    Console_Data.Write(TR);
end Write;      -- to screen

procedure Write(PR : Pressure_Reading) is
begin
    Console_Data.Write(PR);
end Write; -- to screen
end IO;

```

Summary

Process interactions require operating systems and concurrent programming languages to support synchronization and inter-task communication. Communication can be based on either shared variables or message passing. This chapter has been concerned with shared variables, the multiple update difficulties they present and the mutual exclusion synchronizations needed to counter these difficulties. In this discussion, the following terms were introduced:

- **critical section** – code that must be executed under mutual exclusion;
- **producer – consumer system** – two or more tasks exchanging data via a finite buffer;
- **busy waiting** – a task continually checking a condition to see if it is now able to proceed;
- **livelock** – an error condition in which one or more tasks are prohibited from progressing whilst using up processing cycles.

Examples were used to show how difficult it is to program mutual exclusion using only shared variables. Semaphores were introduced to simplify these algorithms and to remove busy waiting. A semaphore is a non-negative integer that can only be acted upon by `wait` and `signal` procedures. The executions of these procedures are atomic.

The provision of a semaphore primitive has the consequence of introducing a new state for a task; namely, **suspended**. It also introduces two new error conditions:

- **deadlock** – a collection of suspended tasks that cannot proceed;
- **indefinite postponement** – a task being unable to proceed as resources are not made available for it (also called **lockout** or **starvation**).

Semaphores can be criticized as being too low-level and error-prone in use. Following their development, five more structured primitives were introduced:

- conditional critical regions
- monitors
- mutexes

- protected objects
- synchronized methods.

Monitors are an important language feature. They consist of a module, entry to which is assured (by definition) to be under mutual exclusion. Within the body of a monitor, a task can suspend itself if the conditions are not appropriate for it to proceed. This suspension is achieved using a condition variable. When a suspended task is awoken (by a `signal` operation on the condition variable), it is imperative that this does not result in two tasks being active in the module at the same time.

A form of monitor can be implemented using a procedural interface. Such a facility is provided by mutexes and condition variables in C/Real-Time POSIX.

Although monitors provide a high-level structure for mutual exclusion, other synchronizations must be programmed using very low-level condition variables. This gives an unfortunate mix of primitives in the language design. Ada's protected objects give the structuring advantages of monitors and the high-level synchronization mechanisms of conditional critical regions.

Integrating concurrency and OOP is fraught with difficulties. Ada tries to simplify the problem by supporting interfaces but not inheritance with protected types. Java, however, addresses the problem by providing synchronized member methods for classes. This facility (along with the synchronized statement and wait and notify primitives) provides a flexible object-oriented based monitor-like facility. Unfortunately, the inheritance anomaly is present with this approach.

The next chapter considers message-based synchronization and communication primitives. Languages that use these have, in effect, elevated the monitor to an active task in its own right. As a task can only be doing one thing at a time, mutual exclusion is assured. Tasks no longer communicate with shared variables but directly. It is therefore possible to construct a single high-level primitive that combines communication and synchronization. This concept was first considered by Conway (1963) and has been employed in high-level real-time programming languages. It forms the basis of the rendezvous in Ada.

Further reading

- Ben-Ari, M. (2005) *Principles of Concurrent and Distributed Programming*. New York: Prentice Hall.
- Burns, A. and Wellings, A. J. (2007) *Concurrent and Real-time Programming in Ada*. Cambridge: Cambridge University Press.
- Butenhof, D. R. (1997) *Programming With Posix Threads*. Reading, MA: Addison-Wesley.
- Goetz, B. (2006) *Java: Concurrency in Practice*. Reading, MA: Addison-Wesley.
- Hyde, P. (1999) *Java Thread Programming*. Indianapolis, IN: Sams Publishing.
- Lea, D. (1999) *Concurrent Programming in Java: Design Principles and Patterns*. Reading, MA: Addison-Wesley.

- Oaks, A. and Wong, H. (1997) *Java Threads*. Sebastopol, CA: O'Reilly.
- Nichols, B., Buttlar, D. and Farrell, J. (1996) *POSIX Threads Programming*. Sebastopol, CA: O'Reilly.
- Wellings, A.J. (2004) *Concurrent and Real-Time Programming in Java*. Chichester: Wiley.

Exercises

- 5.1** Show how Peterson's algorithm given in Section 5.2 can be modified to allow a high-priority task to be given preference over a low-priority one when busy waiting.
- 5.2** Consider a data item which is to be shared between a single producer and a single consumer. The producer is a periodic task which reads a sensor and writes the value to the shared data item. The consumer is a sporadic task which wishes to take the latest value placed in the shared data item by the producer.

The following package claims to provide a generic algorithm with which the producer and consumer can communicate safely, *without* the need for mutual exclusion or busy waiting.

```
generic
  type Data is private;
  Initialvalue : Data;
package Simpsons_Algorithm is
  procedure Write(Item: Data); -- non-blocking
  procedure Read (Item : out Data); -- non-blocking
end Simpsons_Algorithm;

package body Simpsons_Algorithm is

  type Slot is (First, Second);

  Fourslot : array (Slot, Slot) of Data :=
    (First => (Initialvalue, Initialvalue),
     Second => (Initialvalue, Initialvalue));

  Nextslot : array(Slot) of Slot := (First, First);

  Latest : Slot := First;
  Reading : Slot := First;

  procedure Write(Item : Data) is
    Pair, Index : Slot;
  begin
    if Reading = First then
      Pair := Second;
    else
      Pair := First;
    end if;
    if NEXT_SLOT(Pair) = First then
      Index := Second;
    else
```

```

        Index := First;
    end if;
    Fourslot(Pair, Index) := Item;
    Nextslot(Pair) := Index;
    Latest := Pair;
end Write;

procedure Read(Item : out Data) is
    Pair, Index : Slot;
begin
    Pair := Latest;
    Reading := Pair;
    Index := Nextslot(Pair);
    Item := Fourslot(Pair, Index);
end Read;
end Simpsons_Algorithm;

```

Describe carefully how this algorithm works by explaining what happens when:

- (1) a Write is followed by a Read
- (2) a Read is preempted by a Write
- (3) a Write is preempted by a Read
- (4) a Read is preempted by more than one Write
- (6) a Write is preempted by more than one Read

Comment on how the algorithm trades off safe communication against data freshness.

Many compilers optimize code so that frequently accessed variables are held in registers local to a task. In this context, comment on whether the above algorithm's claim of safe communication is justifiable. Are there any changes to the algorithm needed to make it work for all implementations of Ada?

- 5.3 Consider a shared data structure that can be both read from and written to. Show how semaphores can be used to allow many concurrent readers or a single writer, but not both.
- 5.4 Show how Conditional Critical Region can be implemented using semaphores.
- 5.5 Show how Hoare's monitors can be implemented using semaphores.
- 5.6 Show how binary semaphores can be implemented using Hoare's monitors.
- 5.7 One of the criticisms of monitors is that condition synchronization is too low-level and unstructured. Explain what is meant by this statement. A higher level monitor synchronization primitive might take the form

```
WaitUntil boolean_expression;
```

where the task is delayed until the boolean expression evaluates to true. For example:

```
WaitUntil x < y + 5;
```

would delay the task until $x < y + 5$.

Although this form of condition synchronization is more structured it is not found in most languages which support monitors. Explain why this is the case. Under what circumstances would the objections to the above high-level synchronization facility be invalid? Show how the bounded buffer problem can be solved using the `WaitUntil` synchronization primitive inside a monitor.

- 5.8 Consider a system of three cigarette smoker tasks and one agent task. Each smoker continuously makes a cigarette and smokes it. To make a cigarette three ingredients are needed: tobacco, paper and matches. One of the tasks has paper, another tobacco and the third has matches. The agent has an infinite supply of all three ingredients. The agent places two ingredients, chosen randomly, on a table. The smoker who has the remaining ingredient can then make and smoke a cigarette. Once it has finished smoking the smoker signals the agent who then puts another two of the three ingredients on the table, and the cycle repeats.
Sketch the structure of a monitor which will synchronize the agent and all three smokers.
- 5.9 Show how the operations on a semaphore can be implemented in the nucleus of an operating system for a single processor system **without** busy waiting. What hardware facility does your solution require?
- 5.10 Show how C/Real-Time POSIX mutexes and condition variables can be used to implement a shared data structure that can be both read from and written to. Allow many concurrent readers or a single writer but not both.
- 5.11 Show how C/Real-Time POSIX mutexes and condition variables can be used to implement a resource controller.
- 5.12 Complete Exercise 4.8 using Ada's protected objects for task synchronization.
- 5.13 Compare and contrast the facilities provided by the C/Real-Time POSIX mutexes and condition variables with those provided by Ada's protected objects.
- 5.14 Redo Exercise 5.8 using protected objects.
- 5.15 Implement quantity semaphores using protected objects.
- 5.16 Show how one or more protected objects can be used to implement Hoare's monitors.
- 5.17 Explain the synchronization imposed by the following Ada protected object:

```
protected type Barrier(Needed : Positive) is
  entry Wait;
private
  Releasing : Boolean := False;
end Barrier;

protected body Barrier is
  entry Wait when Wait'Count = Needed or Releasing is
  begin
    if Wait'Count = 0 then
      Releasing = False;
    else
      Releasing := True;
    end if;
  end if;
```

```

    end Wait;
end Barrier;

```

The following package provides a simplified Ada binding to Real-Time POSIX Mutexes and Condition Variables. All Mutexes and Condition Variables are initialized with default attributes.

```

package Pthreads is
  type Mutex_T is limited private;
  type Cond_T is limited private;

  procedure Mutex_Initialize (M: in out Mutex_T);
  procedure Mutex_Lock(M: in out Mutex_T);
  procedure Mutex_Trylock(M: in out Mutex_T);
  procedure Mutex_Unlock(M: in out Mutex_T);

  procedure Cond_Initialize(C: in out Cond_T);
  procedure Cond_Wait(C: in out Cond_T;
                      M : in out Mutex_T);

  procedure Cond_Signal(C: in out Cond_T);
  procedure Cond_Broadcast(C: in out Cond_T);
private
  ...
end Pthreads;

```

Show how Barriers, as defined above, can be implemented using this package. Do *not* use any of Ada's communication and synchronization facilities in the solution.

5.18 The following package defines an Ada semaphore abstraction:

```

generic
  Initial : Natural := 1; -- default initial value of semaphore
package Semaphore_Package is
  type Semaphore is limited private;
  procedure Wait (S : Semaphore);
  procedure Signal (S : Semaphore);
private
  type Semaphore is ...; -- not needed for this question
end Semaphore_Package;

```

Using the Semaphore_Package, show how the following communication paradigm (and its associated package specification) can be implemented.

A Multicast is where one task is able to send the same data to several waiting tasks. A package specification for the Multicast abstraction is given below:

```

package Multicast is
  procedure Send(I : Integer);
  procedure Receive(I : out Integer);
end Multicast;

```


The receiver tasks indicate their willingness to receive data by calling the procedure `Receive` defined in the package given above (in this example the data is of type `Integer`). The tasks are blocked by this call. A sender task indicates that it wishes to multicast data by calling the `Send` procedure. All tasks which are currently blocked on the receive are released when the sender invokes the `Send` procedure. The data passed by the sender is given to all waiting tasks. Once the `Send` procedure has finished, any new calls to `Receive` must wait for the next sender.

Show how the body of the `Multicast` package can be implemented using semaphores.

- 5.19** A broadcast is similar to a multicast except that ALL intended recipients must receive the data. A package specification for the `Broadcast` abstraction is given below:

```
package Broadcast is
  -- for 10 tasks
  procedure Send(I : Integer);
  procedure Receive(I : out Integer);
end Broadcast;
```

Consider, for example, a system with 10 recipient tasks. These tasks all call the `Receive` procedure when they are ready to receive the broadcast. The tasks are blocked by this call. A sender task indicates that it wishes to broadcast data by calling the `Send` procedure. The `Send` procedure waits until *all* ten tasks are ready to receive the broadcast before releasing the recipients and passing the data. If more than one call to `Send` occurs, then they are queued.

Show how the body of the `Broadcast` package can be implemented using the semaphore package given in Exercise 5.18.

- 5.20** It has been suggested that York should put a limit on the number of motorists that can enter the city at any one time. One proposal is to establish monitoring checkpoints at each of the city's Bars (entrances through the city's walls) and, when the city is full, to turn the traffic lights to red for incoming traffic. To indicate to the motorists that the city is full, the red light is set to flashing.

In order to achieve this goal, pressure sensors are placed in the road at the Bars' entry and exit points. Every time a car enters the city, a signal is set to a `BarController` task (as a task entry call); similarly when a car exits:

```
Max_Cars_In_City_For_Red_Light : constant Positive := N;
Min_Cars_In_City_For_Green_Light : constant Positive := N - 10;

type Bar is (Walmgate, Goodramgate, Micklegate,
             Bootham, Barbican);

task type Bar_Controller(G : Bar) is
  entry Car_Entered;
  entry Car_Exited;
end Bar_Controller;
```



```

Walmgate_Bar_Controller : Bar_Controller(Walmgate);
Goodramgate_Bar_Controller : Bar_Controller(Goodramgate);
Micklegate_Bar_Controller : Bar_Controller(Micklegate);
Bootham_Bar_Controller : Bar_Controller(Bootham);
Barbican_Bar_Controller : Bar_Controller(Barbican);

```

Show how the body of these tasks can be coordinated so that ONE of them calls the `CityTrafficLightController` (with the following task specification) to indicate whether more cars are to be allowed in or not.

```

task City_Traffic_Lights_Controller is
    entry City_Is_Full;
    entry City_Has_Space;
end City_Traffic_Lights_Controller;

task body Traffic_Lights_Controller is separate;
-- body of no interest in this question

```

- 5.21 Explain why the resource controller given in Section 5.4.7 suffers from a race condition. How can the algorithm be modified to remove this condition?
- 5.22 Show how the reader/writers problem can be implemented in Java where priority is given to readers and where writers are guaranteed to be serviced in a FIFO order.
- 5.23 Show how Java can be used to implement a resource controller.
- 5.24 Implement quantity semaphores using Java.
- 5.25 Consider the following Java class:

```

public class Event {
    public synchronized void highPriorityWait();
    public synchronized void lowPriorityWait();
    public synchronized
    void signalEvent();
}

```

Show how this class be implemented so that `signalEvent` releases one high-priority waiting thread if one is waiting. If there is no high-priority waiting thread, release one low-priority waiting thread. If no thread is waiting, the `signalEvent` has no effect.

Now consider the case where an `Id` can be associated with the methods. How can the algorithm be modified so that the `signalEvent` wakes up the appropriate blocked thread?

Chapter 6

Message-based synchronization and communication

6.1	Process synchronization	6.7	C/Real-Time POSIX
6.2	Task naming and message structure		message queues
6.3	Message passing in Ada	6.8	Distributed systems
6.4	Selective waiting	6.9	Simple embedded system revisited
6.5	The Ada <code>select</code> statement		Summary
6.6	Non-determinism, selective waiting and synchronization primitives		Further reading
			Exercises

The alternative to shared variable synchronization and communication is one based on message passing. The approach is typified by the use of a single construct for both synchronization and communication. Within this broad category, however, a wide variety of language models exist. This variety in the semantics of message passing arises from, and is dominated by, three issues:

- (1) the model of synchronization;
- (2) the method of task naming;
- (3) the message structure.

Each of these issues is considered in turn. Following this, the message passing models of Ada along with the C/Real-time POSIX model are described.

Message passing is the primary method of communication in a distributed system. Hence, this chapter also considers the remote procedure call and remote method invocation in Java and Ada.

6.1 Process synchronization

With all message-based systems, there is the implicit synchronization that a receiver task cannot obtain a message before that message has been sent. Although this is quite obvious, it must be compared with the use of a shared variable; in this case, a receiver task can read a variable and not know whether it has been written to by the sender task. If a task executes an unconditional message receive when no message is available, then it will become suspended until the message arrives.

Variations in the task synchronization model arise from the semantics of the send operation, which can be broadly classified as follows.

- **Asynchronous** (or **no-wait**) – the sender proceeds immediately, regardless of whether the message is received or not.
- **Synchronous** – the sender proceeds only when the message has been received.
- **Remote invocation** – the sender proceeds only when a reply has been returned from the receiver.

To appreciate the difference between these approaches, consider the following analogy. The posting of a letter is an asynchronous send – once the letter has been put into the letter box the sender proceeds with his or her life; only by the return of another letter can the sender ever know that the first letter actually arrived. From the receiver's point of view, a letter can only inform the reader about an out-of-date event; it says nothing about the current position of the sender. (Everyone has received sun-drenched postcards from people they know have been back at work for at least two weeks!)

A telephone is a better analogy for synchronous communication. The sender now waits until contact is made and the identity of the receiver verified before the message is sent. If the receiver can reply immediately (that is, during the same call), the synchronization is remote invocation. Because the sender and receiver 'come together' for a synchronized communication it is often called a **rendezvous**. The remote invocation form is known as an **extended rendezvous**, as arbitrary computations can be undertaken before the reply is sent (that is, during the rendezvous).

Clearly, there is a relationship between these three forms of send. Two asynchronous events can essentially constitute a synchronous relationship if an acknowledgement message is always sent (and waited for):

P1	P2
asyn_send (message)	wait (message)
wait (acknowledgement)	asyn_send (acknowledgement)

Moreover, two synchronous communications can be used to construct a remote invocation:

P1	P2
syn_send (message)	wait (message)
wait (reply)	...
	construct reply
	...
	syn_send (reply)

As an asynchronous send can be used to construct the other two, it could be argued that this model gives the greatest flexibility and should be the one that languages and operating systems adopt. However, there are a number of drawbacks to using this model:

- (1) Potentially infinite buffers are needed to store messages that have not been read yet (perhaps because the receiver has terminated).
- (2) Because asynchronous communication is out-of-date, most sends are programmed to expect an acknowledgement (that is, synchronous communication).

- (3) More communications are needed with the asynchronous model, hence programs are more complex.
- (4) It is more difficult to prove the correctness of the complete system.

Note that where asynchronous communication is desired in a synchronized message passing language, buffer tasks can easily be constructed. However, the implementation of a task is not without cost; therefore, having too many buffer tasks might have a detrimental effect on the overall performance of the system.

6.2 Task naming and message structure

Task naming involves two distinct sub-issues: direction versus indirection, and symmetry. In a direct naming scheme, the sender of a message explicitly names the receiver:

```
send <message> to <task-name>
```

With an indirect naming scheme, the sender names some intermediate entity (known variously as a channel, mailbox, link or pipe):

```
send <message> to <mailbox>
```

Note that, even with a mailbox, the message passing can still be synchronous (that is, the sender will wait until the message is read). Direct naming has the advantage of simplicity, while indirect naming aids the decomposition of the software; a mailbox can be seen as an interface between distinct parts of the program.

A naming scheme is symmetric if both sender and receiver name each other (directly or indirectly):

```
send <message> to <task-name>
wait <message> from <task-name>
```

```
send <message> to <mailbox>
wait <message> from <mailbox>
```

It is asymmetric if the receiver names no specific source but accepts messages from any task (or mailbox):

```
wait <message>
```

Asymmetric naming fits the client-server paradigm in which 'server' tasks provide services in response to messages from any of a number of 'client' tasks. An implementation, therefore, must be able to support a queue of tasks waiting for the server.

If the naming is indirect then there are further issues to consider. The intermediary could have:

- a many-to-one structure (that is, any number of tasks could write to it but only one task can read from it); this again fits the client-server paradigm;
- a many-to-many structure (that is, many clients and many servers);
- a one-to-one structure (that is, one client and one server); note that with this structure no queues need to be maintained by the run-time support system;

- a one-to-many structure; this situation is useful when a task wishes to send a request to a group of worker tasks and it does not care which task services the request.

6.2.1 Message structure

Ideally a language should allow any data object of any defined type (predefined or user) to be transmitted in a message. Living up to this ideal is difficult, particularly if data objects have different representations at the sender and receiver, and even more so if the representation includes pointers.

6.3 Message passing in Ada

Ada, as well as supporting communication through shared variables and protected types, allows communication and synchronization to be based on message passing.

6.3.1 The Ada model

The semantics of remote invocation have many superficial similarities with a procedure call. Data passes to a receiver, the receiver executes and then data is returned. Because of this similarity, Ada supports the definition of a program's messages in a way that is compatible with the definition of procedures, protected subprograms and entries. In particular, the parameter-passing models are identical (that is, there is only one model that is used in all situations).

In order for a task to receive a message, it must define an **entry**. As before, any number of parameters, of any mode and of any type are allowed. For example:

```
task type Screen_Output (Id : Screen_Identifier) is
  -- a task type definition
  entry Call (Value : Character; X_Coordinate,
             Y_Coordinate: Integer);
end Screen_Output;
```

```
Display: Screen_Output (Tty1);
-- where Tty1 is of type Screen_Identifier
```

```
task Time_Server is -- a single task definition
  entry Read_Time (Now : out Time);
  entry Set_Time (New_Time : Time);
end Time_Server;
```

Entries may be defined as private; this means that they can only be called by tasks local to the task's body. For example, consider the following `Telephone_Operator` task type. It provides three services: an enquiry entry requiring the name and address of a subscriber, an alternative enquiry entry requiring the name and postal code of a subscriber and a fault-reporting service requiring the number of the faulty line. The task also has a private entry for use by its internal tasks:

```
task type Telephone_Operator is
  entry Directory_Enquiry (Person : in Name; Addr : in Address;
                        Num : out Number);
```



```

entry Directory_Enquiry(Person : in Name;
    Zip : in Postal_Code; Num : out Number);

entry Report_Fault(Num : Number);
private
    entry Allocate_Repair_Worker(Num : out Number);
end Telephone_Operator;

```

Ada also provides a facility whereby an array of entries can, in effect, be defined – these are known as **entry families**. For instance, consider a multiplexor which has seven input channels. Rather than representing each of these as a separate entry, Ada allows them to be defined as a family.¹

```

type Channel_Number is new Integer range 1 .. 7;
task Multiplexor is
    entry Channels(Channel_Number)(Data: Input_Data);
end Multiplexor;

```

The above defines seven entries, all with the same parameter specification.

To call a task (that is, send it a message) simply involves naming the receiver task and its entry (naming is direct); for example:

```

Display.Call(Char,10,20); -- where Char is a character

Multiplexor.Channels(3)(D);
-- where 3 indicates the index into the entry family
-- and D is of type Input_Data

Time_Server.Read_Time(T); -- where T is of type Time

```

Note that, in the last example, the only data being transferred is passing in the opposite direction to the message itself (via an ‘out’ parameter). This can lead to terminology confusion, and hence the term ‘message passing’ is not usually applied to Ada. The phrase ‘extended rendezvous’ is less ambiguous.

Ada’s protected entries and task entries are identical from the calling task’s perspective.

If an entry call is made on a task that is no longer active then the exception `Tasking_Error` is raised at the point of the call. This allows alternative action to be taken if, unexpectedly, a task has terminated prematurely, as illustrated below:

```

begin
    Display.Call(C,I,J);
exception
    when Tasking_Error => -- log error and continue
end;

```

Note that this is not equivalent to checking beforehand that the task is available:

```

if Display'Terminated then
    -- log error and continue

```

¹Ada also allows families of protected entries and protected private entries.

```

else
  Display.Call(C,I,J);
end if;

```

An interleaving could cause a task to terminate after the attribute has been evaluated but before the call is handled.

To receive a message involves accepting a call to the appropriate entry:

```

accept Call(C: Character; I,J : Integer) do
  Local_Array(I,J) := C;
end Call;

accept Read_Time(Now : out Time) do
  Now := Clock;
end Read_Time;

accept Channels(3)(Data: Input_Data) do
  -- store data from the 3rd channel in the family
end Channels;

```

An **accept** statement must be in a task body (not a called procedure) but can be placed where any other statement is valid. It can even be placed within another accept statement (although not for the same entry). All entries (and family members) should have accepts associated with them. These accepts name the entry concerned but not the task from which a call is sought. Naming is thus asymmetric. A number of tasks can call the same entry and will be queued until they are accepted. The queue is serviced in a FIFO order; however, the Real-Time Systems Annex provides a mechanism to allow the queue to be priority ordered.

To give a simple example of tasks interacting, consider, two tasks that loop round and pass data between them. In the Ada code, the tasks will swap data:

```

procedure Test is
  Number_Of_Exchanges : constant Integer := 1000;

  task T1 is
    entry Exchange (I : Integer; J : out Integer);
  end T1;

  task T2;

  task body T1 is
    A,B : Integer;
  begin
    for K in 1 .. Number_Of_Exchanges loop
      -- produce A
      accept Exchange (I : Integer; J : out Integer) do
        J := A;
        B := I;
      end Exchange;
      -- consume B
    end loop;
  end T1;

```



```

task body T2 is
  C,D : Integer;
begin
  for K in 1 .. Number_Of_Exchanges loop
    -- produce C
    T1.Exchange(C,D);
    -- consume D
  end loop;
end T2;
begin
  null;
end Test;

```

Although the relationship between the two tasks (T1 and T2) is essentially symmetric, the asymmetric naming in Ada requires them to have quite different forms.

6.3.2 Exception handling and the rendezvous

As any valid Ada code can be executed during a rendezvous, there is the possibility that an exception could be raised within the **accept** statement itself. If this occurs then either:

- there is a valid exception handler within the accept (either as part of the accept statement, or within a block nested within the accept statement); in which case the accept will terminate normally; or
- the raised exception is not handled within the accept and the accept is immediately terminated.

In this latter case, the named exception will be re-raised in both the called and calling tasks. The called task will have the exception raised immediately after the accept; the calling task will have it raised after the entry call. Scope problems may, however, cause the exception to be anonymous in the calling task.

To illustrate the interaction between the rendezvous and the exception models, consider a task that acts as a file server. One of its entries will allow a client task to open a file:

```

task File_Handler is
  entry Open(F : File_Type);
  ...
end File_Handler;

task body File_Handler is
  ...
begin
  loop
    begin
      ...
      accept Open(F : File_Type) do
        loop
          begin
            Device_Open(F);

```

```

        return;
    exception
        when Device_Off_Line =>
            Boot_Device;
        end;
    end loop;
end Open;
...
exception
    when File_Does_Not_Exist =>
        null;
    end;
end loop;
end File_Handler;

```

In this code, the `File_Handler` calls a device driver to open the specified file. This request can either succeed or lead to one of two exceptions being raised: `Device_Off_Line` or `File_Does_Not_Exist`. The first exception is handled within the `accept`; an attempt is made to boot the device and then the open request is repeated. As the exception handler is within the `accept`, the client is unaware of this activity (although if the device refuses to boot, it will be indefinitely suspended). The second exception is due to a faulty request by the user task. It is therefore not handled within the `accept` and will propagate out to the calling task which will need to protect itself against the exception:

```

begin
    File_Handler.Open(New_File);
exception
    when File_Does_Not_Exist =>
        File_Handler.Create(New_File);
        File_Handler.Open(New_File);
end;

```

Note that the server task also protects itself against this exception by having a block defined within the outer loop construct.

6.3.3 Message passing via task interfaces

An Ada task specification defines how other tasks can communicate with it. Ada's direct asymmetric naming means that client tasks must explicitly name the server. In some situations this is not appropriate. The clients may not want to be concerned with the type of the task that will service its requests, only that it provides a particular service. Task interfaces are ideal for this purpose.

A task type in Ada 2005 can be declared to 'implement' zero, one or more combinations of synchronized (see Section 5.8.2) and task interfaces. So for example, consider the following interface declaration:

```

package Simple_Task_Interface is
    type Simple_TI is task interface;
    procedure OP1(TI : in out Simple_TI) is abstract;
    type Any_Simple_TI is access all Simple_TI'Class;
end Simple_Task_Interface;

```


An object of the access type can reference any task that implements this interface. For example, objects of the following task types can be referred to:

```
task type Example_Task_Type_1 is new Simple_TI with
  overriding entry OP1;
end Example_Task_Type_1;

task type Example_Task_Type_2 is new Simple_TI with
  overriding entry OP1;
  entry OP2;
end Example_Task_Type_2;
```

Note that the task interface contains subprogram declarations even though the task implements them as entries. It is not possible to put an entry declaration in an interface definition.

Consider now the following interface and task:

```
type Another_TI is task interface and Simple_TI;
procedure OP2(TI : in out Another_TI) is abstract;
type Any_Another_TI is access all Another_TI'Class;

task type Example_Task_Type_3 is new Another_TI with
  overriding entry OP1;
end Example_Task_Type_3;
overriding procedure OP2(TI : in out Example_Task_Type_3);
```

In this example, the `Another_TI` type inherits the subprogram defined in the `Simple_TI` interface and adds the new subprogram `OP2`. The task `Example_Task_Type_3` implements this interface but only provides an entry for `OP1`. `OP2` is provided as a procedure. The aim here is for `OPP` to provide some pre- or post-processing before or after accessing the task. For example:

```
procedure OP2(TI : in out Example_Task_Type_3) is
begin
  if not TI'Terminated then
    TI.OP1;
    Ada.Text_IO.Put_line("Task called succesfully");
  else
    Ada.Text_IO.Put_line("Task already terminated");
  end if;
end OP2;
```

Note, it is not possible to declare a task type that is extended from another task type. A task type can only be derived from one or more interfaces.

6.4 Selective waiting

In all the forms of message passing that have so far been discussed, the receiver of a message must wait until the specified task, or channel, delivers the communication. This is, in general, too restrictive. A receiver task may actually wish to wait for any one of a number of tasks to call it. Server tasks receive request messages from a number of clients, the order in which the clients call being unknown to the servers. To facilitate this common program structure, receiver tasks are allowed to wait selectively for a

number of possible messages. But to understand selective waiting fully, Dijkstra's guarded commands (Dijkstra, 1975) must be explained first.

A guarded command is one which is executed only if its guard evaluates to TRUE. For example:

```
x < y -> m := x
```

This means that if x is less than y then assign the value of x to m . A guarded command is not a statement in itself, but is a component of a **guarded command set**, of which there are a number. Here, the concern is only with the choice, or alternative, construct:

```
if x <= y -> m := x
□ x >= y -> m := y
fi
```

The \square signifies choice. In the above example, the program's execution will either assign x to m or y to m . If both alternatives are possible, that is, both guards evaluate true ($x = y$ in this example), then an *arbitrary* choice is made. The programmer cannot determine which path will be taken, the construct is **non-deterministic**. A well-constructed program will be valid for all possible choices. When $x = y$, in this example, both paths will have the same effect.

It is important to note that this non-deterministic structure is quite distinct from the deterministic form that could have been constructed using the normal `if` statement:

```
if x <= y then m := x;
elseif x >= y then m := y;
end if;
```

Here, the values $x = y$ would ensure that m was assigned the value x .

The general choice construct can have any number of guarded components. If more than one guard evaluates to TRUE, the choice is arbitrary. But if no guard evaluates to TRUE then this is viewed to be an error condition and the statement, along with the task that executed it, is aborted.

The guarded command is a general program structure. If, however, the command being guarded is a message operator (normally the receive operator, although in some languages also the send), then the statement is known as **selective waiting**. This was first introduced in CSP (Hoare, 1978) and is available in Ada.

6.5 The Ada `select` statement

Ada's many-to-one message-passing relationship can deal easily with a number of clients by having them all call the same entry. However, where a server must deal with possible calls to two or more different entries, selective waiting is required; in Ada this is provided by the **select** statement. Consider, for illustration, a server task which offers two services, via entries $S1$ and $S2$. The following structure is often sufficient (that is, a loop containing a **select** statement which offers both services):

```
task Server is
  entry S1(...);
  entry S2(...);
end Server;
```



```

task body Server is
    ...
begin
    loop
        -- prepare for service
        select
            accept S1(...) do
                -- code for this service
            end S1;
        or
            accept S2(...) do
                -- code for this service
            end S2;
        end select;
        -- do any housekeeping
    end loop;
end Server;

```

On each execution of the loop, one of the **accept** statements will be executed.

This Ada program does not illustrate the use of boolean expressions in guards. The general form of the Ada select is:

```

select
    when <Boolean-Expression> =>
        accept <entry> do
            ..
        end <entry>;
        -- any sequence of statements
or
    -- similar
    ...
end select;

```

There can be any number of alternatives. Apart from accept alternatives (of which there must be at least one) there are three further forms (which cannot be mixed in the same statement):

- (1) a terminate alternative;
- (2) an else alternative;
- (3) a delay alternative.

The delay alternative is considered in Chapter 9. The else alternative is defined to be executed when (and only when) no other alternative is *immediately* executable. This can only occur when there are no outstanding calls on entries which have boolean expressions that evaluate True (or no boolean expressions at all).

The terminate alternative has the following important properties.

- (1) If it is chosen, the task that executed the select is finalized and terminated.
- (2) It can only be chosen if there are no longer any other tasks that can call the entries of this task.

To be more precise, a task will terminate if all tasks that are dependent on the same master have already terminated or are similarly waiting on select statements with terminate alternatives. The effect of this alternative is to allow server tasks to be constructed that need not concern themselves with termination, but will nevertheless terminate when they are no longer needed.

The execution of the select statement performs the following actions. First the boolean expressions are evaluated; those that produce a false value lead to that alternative being closed for that execution of the select. Following this phase, a collection of possible alternatives is derived. If this collection is empty, the exception `Program_Error` is raised immediately after the select. For normal execution, one alternative is chosen. The choice is arbitrary if there is more than one alternative with an outstanding call. If there are no outstanding calls on eligible alternatives, either:

- the else alternative is executed, if there is one;
- the task is suspended waiting for a call to be made (or a timeout to expire – see Chapter 9);
- the task is terminated if there is a terminate option *and* there are no other tasks that could call it (as described above).

Note that shared variables can be contained in guards, but this is not recommended as changes to them will not be noticed until the guard is re-evaluated. Also, the Real-Time Systems Annex allows priority to be given to the alternatives of the select according to the textual ordering.

As a final example of the Ada `select` statement, consider the body of the `Telephone_Operator` task given in Section 6.3.1:

```
task body Telephone_Operator is
  Workers : constant Integer := 10;
  Failed : Number;
  task type Repair_Worker;
  Work_Force : array (1 .. Workers) of Repair_Worker;
  task body Repair_Worker is ...;
  ...
begin
  loop
    -- prepare to accept next request
    select
      accept Directory_Enquiry(Person : in Name;
                               Addr : in Address; Num : out Number) do
        -- look up telephone number and
        -- assign the value to Num
      end Directory_Enquiry;
    or
      accept Directory_Enquiry(Person : in Name;
                               Zip : in Postal_Code; Num : out Number) do
        -- look up telephone number and
        -- assign the value to Num
      end Directory_Enquiry;
    or
      accept Report_Fault(Num : Number) do
```



```

    Failed := Num;
  end Report_Fault;
  -- store failed number
or
  when Unallocated_Faults =>
    accept Allocate_Repair_Worker(Num : out Number) do
      -- get next failed number
      Num := ...;
    end Allocate_Repair_Worker;
    -- update record of failed unallocated numbers
or
  terminate;
end select;
...
end loop;
end Telephone_Operator;

```

A local task type `Repair_Worker` is responsible for repairing line faults when they are logged. They communicate with the `Telephone_Operator` via the `Allocate_Repair_Worker` private entry. To ensure that the worker tasks do not continually communicate with the `Telephone_Operator`, the accepted statement is guarded. Note also, the `Telephone_Operator` carries out as much work as possible outside the rendezvous to enable the client tasks to continue as quickly as possible.

An Ada `select` statement can also be used by a client task (see Section 9.4.2) and to handle asynchronous events (see Section 7.6.1).

6.6 Non-determinism, selective waiting and synchronization primitives

In the above discussion, it was noted that when there is more than one ready alternative in a selective waiting construct, the choice between them is arbitrary. The rationale behind making the select arbitrary is that concurrent languages usually make few assumptions about the order in which tasks are executed. The scheduler is assumed to schedule tasks non-deterministically (although individual schedulers will have deterministic behaviour).

To illustrate this relationship, consider a task `P` that will execute a selective wait construct upon which tasks `S` and `T` could call. If the scheduler's behaviour is assumed to be non-deterministic then there are a number of possible interleavings or 'histories' for this program.

- (1) `P` runs first; it is blocked on the select statement. `S` (or `T`) then runs and rendezvous with `P`.
- (2) `S` (or `T`) runs first and blocks on the call to `P`; `P` now runs and executes the select statement with the result that a rendezvous takes place with `S` (or `T`).
- (3) `S` (or `T`) runs first and blocks on the call to `P`; `T` (or `S`) now runs and is also blocked on `P`. Finally `P` runs and executes the select statement on which `T` and `S` are waiting.

These three possible and legal interleavings lead to `P` having either none, one or two calls outstanding on the selective wait. The select is defined to be arbitrary precisely because

the scheduler is assumed to be non-deterministic. If P, S and T can execute in any order then, in case (3), P should be able to choose to rendezvous with S or T – it will not affect the program's correctness.

A similar argument applies to any queue that a synchronization primitive defines. Non-deterministic scheduling implies that all such queues should release tasks in a non-deterministic order. Although semaphore queues are often defined in this way, entry queues and monitor queues are usually specified to be FIFO. The rationale here is that FIFO queues prohibit starvation. This argument is, however, spurious; if the scheduler is non-deterministic then starvation can occur (a task may never be given a processor). It is inappropriate for the synchronization primitive to attempt to prevent starvation. It is arguable that entry queues should also be non-deterministic.

The scheduling of tasks which have priorities assigned is considered in detail in Chapter 11.

6.7 C/Real-Time POSIX message queues

C/Real-Time POSIX supports asynchronous, indirect message passing through the notion of message queues. A message queue can have many readers and many writers. Priority may be associated with each message (see Section 12.6).

Message queues are really intended for communication between processes. However, there is nothing to stop their use between threads in the same process, although it is more efficient to use shared memory and mutexes for this purpose (see Section 5.7).

All message queues have attributes which indicate the maximum size of the queue, the maximum size of each message in the queue, the number of messages currently queued and so on. An attribute object is used to set the queue attributes when it is created. The attributes of a queue are manipulated by the `mq_getattr` and `mq_setattr` functions (these functions manipulate the attributes themselves, not an attribute object; this is different from the thread or mutex attributes).

Message queues are given a name when they are created (similar to a file name but not necessarily represented in the file system). To gain access to the queue, simply requires the user process to `mq_open` the associated name. As with all Unix-like file systems, `mq_open` is used to both create and open an already existing queue. (There are also corresponding `mq_close` and `mq_unlink` routines.)

Sending and receiving messages is done via the `mq_send` and `mq_receive` routines. The data is read/written from/to a character buffer. If the buffer is full or empty, the sending/receiving process is blocked unless the attribute `O_NONBLOCK` has been set for the queue (in which case an error return is given). If senders and receivers are waiting when a message queue becomes unblocked, it is not specified which one is woken up unless the priority scheduling option is specified. If the process is multithreaded, each thread is considered to be a potential sender/receiver in its own right.

A process can also indicate that a signal (see Section 7.5.1) should be sent to it when an empty queue receives a message and there are no waiting receivers. In this way, a process can continue executing while waiting for messages to arrive on one or more message queues. It is also possible for a process to wait for a signal to arrive. This allows the equivalent of selective waiting to be implemented.

Program 6.1 summarizes a typical C interface to the Real-Time POSIX message-passing facility.

Program 6.1 The C/Real-Time POSIX interface to message queues.

```

typedef ... mqd_t;
typedef ... mode_t;
typedef ... size_t;
typedef ... ssize_t;

struct mq_attr {
    ...
    long mq_flags;
    long mq_maxmsg;
    long mq_msgsize;
    long mq_curmsg;
    ...
};

/* definitions for O_CREAT, O_EXCL, O_NONBLOCK, O_RDONLY,
   O_WRONLY, O_RDWR */

int mq_getattr(mqd_t mq, struct mq_attr *attrbuf);
    /* get the current attributes associated with mq */
int mq_setattr(mqd_t mq, const struct mq_attr *new_attr,
               struct mq_attr *old_attr);
    /* set the current attributes associated with mq */

mqd_t mq_open(const char *mq_name, int oflags, mode_t mode,
              struct mq_attr *mq_attr);
    /* open/create the named message queue */

int mq_close(mqd_t mq);
    /* close the message queue */

int mq_unlink(const char *mq_name);

ssize_t mq_receive(mqd_t mq, char *msg_buffer,
                  size_t buflen, unsigned int *msgprio);
    /* get the next message in the queue and store it in the */
    /* area pointed at by msg_buffer; */
    /* the actual size of the message is returned */
ssize_t mq_timedreceive(mqd_t mq, char *msg_buffer,
                       size_t buflen, unsigned int *msgprio,
                       const struct timespec *abs_timeout);
    /* as for mq_receive but with a timeout */
    /* returns ETIMEDOUT if the timeout expires */

int mq_send(mqd_t mq, const char *msg,
            size_t msglen, unsigned int msgprio);
    /* send the message pointed at by msg */

int mq_timedsend(mqd_t mq, const char *msg,
                 size_t msglen, unsigned int msgprio,
                 const struct timespec *abs_timeout);
    /* send the message pointed at by msg with a timeout */
    /* returns ETIMEDOUT if the timeout expires */

```

```

int mq_notify(mqd_t mq, const struct sigevent *notification);
    /* request that a signal be sent to the calling process */
    /* if a message arrives on an empty mq and there are no */
    /* waiting receivers */

/* All the above integer functions return 0 if successful, else -1. */
/* When an error condition is returned by any of the above functions, */
/* a shared variable errno contains the reason for the error */

```

To illustrate the use of message queues, the simple robot arm discussed in Chapters 4 and 5 is sketched with a parent process forking three controllers. This time, the parent communicates with the controllers to pass them the new position of the arm. First the controller code is given. Here, `MQ_OPEN` and `FORK` are assumed to be functions which test the error return from the `mq_open` and `fork` system calls and only return if the calls are successful.

```

typedef enum xplane, yplane, zplane dimension;

void move_arm(dimension D, int P);

#define DEFAULT_NBYTES 4
/* assume that the coordinate can be represented as 4 characters */
int nbytes = DEFAULT_NBYTES;

#define MQ_XPLANE    "/mq_xplane" /* message queue name */
#define MQ_YPLANE    "/mq_yplane" /* message queue name */
#define MQ_ZPLANE    "/mq_zplane" /* message queue name */
#define MODE ...      /* mode information for mq_open */
/* names of message queues */

void controller(dimension dim) {
    int position, setting;
    mqd_t my_queue;
    struct mq_attr ma;
    char buf[DEFAULT_NBYTES];
    ssize_t len;

    position = 0;
    switch(dim) { /* open appropriate message queue */
        case xplane:
            my_queue = MQ_OPEN(MQ_XPLANE, O_RDONLY, MODE, &ma);
            break;
        case yplane:
            my_queue = MQ_OPEN(MQ_YPLANE, O_RDONLY, MODE, &ma);
            break;
        case zplane:
            my_queue = MQ_OPEN(MQ_ZPLANE, O_RDONLY, MODE, &ma);
            break;
        default:
            return;
    };
}

```



```

while (1) {
    /* read message */
    len = MQ_RECEIVE(my_queue, &buf[0], nbytes, NULL);
    setting = *((int *) (&buf[0]));
    position = position + setting;
    move_arm(dim, position);
};
}

```

Now the main program which creates the controller processes and passes the appropriate coordinates to them can be given:

```

int main(int argc, char **argv) {
    mqd_t mq_xplane, mq_yplane, mq_zplane;
    /* one queue for each process */
    struct mq_attr ma; /* queue attributes */
    int xpid, ypid, zpid;
    char buf[DEFAULT_NBYTES];

    /* set the required message queue attributes */
    ma.mq_flags = 0; /* No special behaviour */
    ma.mq_maxmsg = 1;
    ma.mq_msgsize = nbytes;

    /* calls to set the actual attributes for the */
    /* three message queues */

    mq_xplane = MQ_OPEN(MQ_XPLANE, O_CREAT|O_EXCL, MODE, &ma);
    mq_yplane = MQ_OPEN(MQ_YPLANE, O_CREAT|O_EXCL, MODE, &ma);
    mq_zplane = MQ_OPEN(MQ_ZPLANE, O_CREAT|O_EXCL, MODE, &ma);

    /* Duplicate the process to get the three controllers */
    switch (xpid = FORK()) {
        case 0: /* child */
            controller(xplane);
            exit(0);
        default: /* parent */
            switch (ypid = FORK()) {
                case 0: /* child */
                    controller(yplane);
                    exit(0);
                default: /* parent */
                    switch (zpid = FORK()) {
                        case 0: /* child */
                            controller(zplane);
                            exit(0);
                        default: /* parent */
                            break;
                    }
            }
    }
}

```

```

while (1) {
    /* find new position and set up buffer to transmit each

```

```

        coordinate to the controllers, for example */
    MQ_SEND(mq_xplane, &buf[0], nbytes, 0);
}
}

```

6.8 Distributed systems

Where tasks execute on different nodes in a distributed system, the goal is to make communication between them as easy and reliable as possible. Unfortunately, in reality, communication often takes place between heterogeneous processors across an unreliable network, and in practice complex communication protocols are required. What is needed is to provide mechanisms whereby:

- Tasks do not have to deal with the underlying form of messages. For example, they do not need to translate data into bit strings suitable for transmission or to break up the message into packets.
- All messages received by user tasks can be assumed to be intact and in good condition. For example, if messages are broken into packets, the run-time system will only deliver them when all packets have arrived at the receiving node and can be properly reassembled. Furthermore, if the bits in a message have been scrambled, the message either is not delivered or is reconstructed before delivery; clearly some redundant information is required for error checking and correction.
- Messages received by a task are the kind that the task expects. The task does not need to perform run-time checks.
- Tasks are not restricted to communication only in terms of a predefined built-in set of types. Instead, tasks can communicate in terms of values of interest to the application. Ideally, if the application is defined using abstract data types or classes, then values of these types can be communicated in messages.

It is possible to identify three main *de facto* standards by which distributed programs can communicate with each other:

- by using an externally-defined application programmer's interface (API), such as *sockets*, to access network transport protocols;
- by using the remote procedure call (RPC) paradigm;
- by using the distributed object paradigm.

The remainder of this subsection will consider RPCs and distributed objects including the Common Object Request Broker Architecture (CORBA).

6.8.1 Remote procedure calls

The overriding goal behind the remote procedure call paradigm is to make distributed communication as simply as possible. Typically, RPCs are used for communication between programs written in the same language, for example, Ada or Java. A procedure (server) is identified as being one that can be called remotely. From the server specification, it is possible to generate automatically two further procedures: a **client stub**

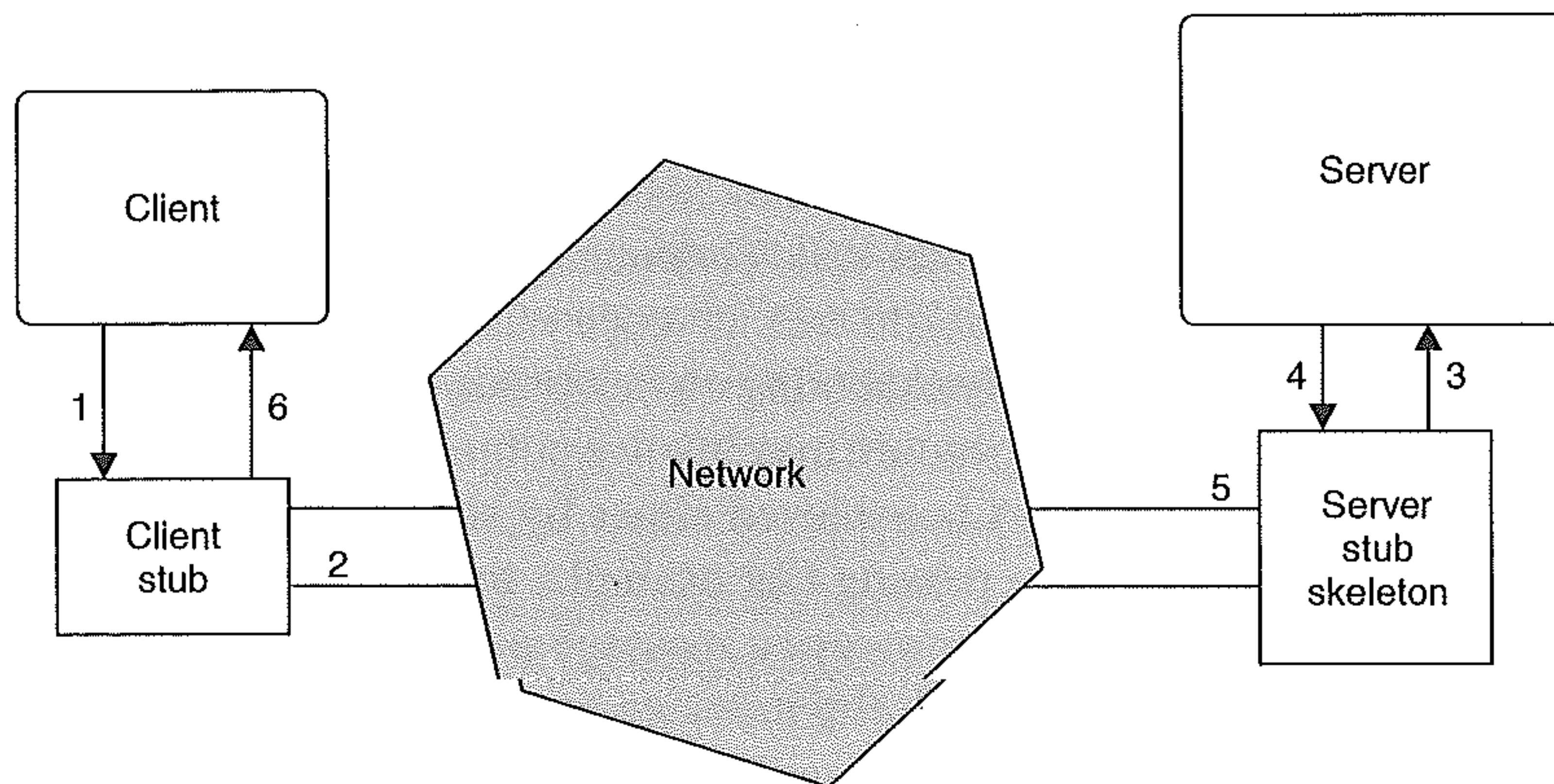


Figure 6.1 The relationship between client and server in an RPC.

and a **server stub**. The client stub is used in place of the server at the site on which the remote procedure call originates. The server stub is used on the same site as the server procedure. The purpose of these two procedures is to provide the link between the client and the server in a transparent way (and thereby meet all the requirements laid out above). Figure 6.1 illustrates the sequence of events in a RPC between a client and a server via the two stub procedures. The stubs are sometimes called **middleware** as they sit between the application and the operating system. The role of the client stub is to:

- identify the address of the server (stub) procedure;
- convert the parameters of the remote call into a block of bytes suitable for transmission across the network – this activity is often call **parameter marshalling**;
- send the request to execute the procedure to the server (stub);
- wait for the reply from the server (stub) and unmarshal the parameters or any exceptions propagated;
- return control to the client procedure along with the returned parameters, or raise an exception in the client procedure.

The goal of the server stub is to:

- receive requests from client (stub) procedures;
- unmarshal the parameters;
- call the server;
- catch any exceptions that are raised by the server;
- marshal the return parameters (or exceptions) suitable for transmission across the network;
- send the reply to the client (stub).

Where the client and server procedures are written in different languages or are on different machine architectures, the parameter marshalling and unmarshalling mechanisms will convert the data into a machine- and language-independent format (see Section 6.8.5).

Remote procedure calls and remote invocation

In a single-processor system, tasks execute procedures in order to transfer control from one section of code to another; only when they need to communicate and synchronize with another task do they require inter-task communication. Remote procedure calls extend this idea to enable a single task to execute code which resides on more than one machine. They allow a task currently executing on one processor to execute a procedure on another. The execution of the procedure may involve communication with tasks which reside on the remote machine; this is achieved either by shared variable methods (for example, monitors) or by message passing (for example, rendezvous).

It is worth noting that remote invocation message passing can be made to appear syntactically like a procedure call, with the message encoded in the input parameters and the reply encoded in the output parameters (see, for example, Ada entries and accepts). However, this syntactic convenience can be misleading, since the semantics of remote invocation message passing are quite different from those of a procedure call. In particular, remote invocation relies on the active cooperation of the receiver task in executing an explicit receive operation. A procedure call, on the other hand, is not a form of inter-task communication, but a transfer of control to a passive piece of code. When the procedure is local (on the same machine as the caller), its body can be executed by the calling task; in the case of a remote procedure call, the body may have to be executed on behalf of the caller by an anonymous surrogate task on the remote machine (which executes the server stub). The involvement of another task in this case is a matter of implementation, not of semantics. For a remote procedure call to have the same semantics as a (re-entrant) local one, the implementation must allow an arbitrary number of calls to be handled concurrently. This requires a task/thread to be created for each call, or the existence of a pool of tasks/threads large enough to handle the maximum number of concurrent calls. The cost of task creation or maintenance may sometimes dictate that the degree of concurrency be limited.

6.8.2 The distributed object model

The term **distributed objects** (or **remote objects**) has been used over the last few years in a variety of contexts. In its most general sense, the distributed object model allows:

- the dynamic creation of an object (in any language) on a remote machine;
- the identification of an object to be determined and held on any machine;
- the transparent invocation of a remote method in an object as if it were a local method and irrespective of the language in which the object is written;
- the transparent run-time dispatching of a method call across the network.

Not all systems which support distributed objects provide mechanisms to support all this functionality. As will be shown in the following subsections:

- **Ada** – supports the static allocation of objects, allows the identification of remote Ada objects, facilitates the transparent execution of remote methods, and supports distributed run-time dispatching of method calls;

- **Java** – allows the code of a Java object to be sent across the network and instances to be created remotely, the remote naming of a Java object, the transparent invocation of its methods, and distributed run-time dispatching;
- **CORBA** – allows objects to be created in different languages on different machines, facilitates the transparent execution of remote methods, and supports distributed run-time dispatching of method calls.

6.8.3 Ada

Ada's model of communication between partitions in a distributed program is via remote subprogram (procedures or functions) calls. A `Remote_Call_Interface` package defines the interface between active partitions. Its body exists only within a single partition. All other occurrences will have library stubs allocated.

The specification of a `Remote_Call_Interface` package must be preelaborable; in addition other restrictions apply, for example it must not contain the definition of a variable (to ensure no remote data access).

A package which is not categorized by any categorization pragma is called a *normal* library package. If it is included in more than one partition, then it is replicated and all types and objects are viewed as distinct.

The above pragmas facilitate the distribution of an Ada program and ensure that illegal partitioning (which allows direct remote variable access between partitions) is easily identifiable.

There are three different ways in which a calling partition can issue a remote subprogram call:

- by calling a subprogram which has been declared in a remote call interface package of another partition directly;
- by dereferencing a pointer to a remote subprogram;
- by using run-time dispatching to a method of a remote object.

It is important to note that, in the first type of communication, the calling and the called partitions are statically bound at compile-time. However, in the latter two, the partitions are dynamically bound at run-time. Hence Ada can support transparent access to resources. As a simple example of the first approach, consider a server that wishes to return details of the local weather forecast for its location. An appropriate interface might be:

```
package Weather_Forecast is
  pragma Remote_Call_Interface;
  function Get_Forecast return String;
end Weather_Forecast;
```

The body of the package would interact with the rest of the program to obtain the forecast, for example, the Yorkshire server in the UK might be written as:

```
package body Weather_Forecast is
  function Get_Forecast return String is
```



```

begin
    return "Sunny";
end Get_Forecast;
end Weather_Forecast;

```

Many remote calls contain only 'in' or 'access' parameters (that is, data that is being passed in the same direction as the call) and a caller may wish to continue its execution as soon as possible. In these situations it is sometimes appropriate to designate the call as an *asynchronous* call. Whether a procedure is to be called synchronously or asynchronously is considered by Ada to be a property of the procedure and not of the call. This is indicated by using a pragma `Asynchronous` when the procedure is declared.

Ada has defined how distributed programs can be partitioned and what forms of remote communication must be supported. However, the language designers were keen not to overspecify the language and not to prescribe a distributed run-time support system for Ada programs. They wanted to allow implementors to provide their own network communication protocols and, where appropriate, allow other ISO standards to be used; for example the ISO Remote Procedure Call standard. To achieve these aims, the Ada language assumes the existence of a standard implementation-provided subsystem for handling all remote communication (the Partition Communication Subsystem, PCS). This allows compilers to generate calls to a standard interface without being concerned with the underlying implementation. The package defined in Program 6.2 illustrates the interface to the remote procedure (subprogram) call (RPC) support system which is part of the PCS.

The type `Partition_Id` is used to identify partitions. For any library-level declaration, `D`, `D'Partition_Id` yields the identifier of the partition in which the declaration was elaborated. The exception `Communication_Error` is raised when an error is detected by `System.RPC` during a remote procedure call. An object of stream type `Params_Stream_Type` is used for marshalling (translating data into an appropriate stream-oriented form) and unmarshalling the parameters or results of a remote subprogram call, for the purposes of sending them between partitions. The object is also used to identify the particular subprogram in the called partition.

The procedure `Do_RPC` is invoked by the calling stub after the parameters are flattened into the message. After sending the message to the remote partition, it suspends the calling task until a reply arrives. The procedure `Do_APC` acts like `Do_RPC` except that it returns immediately after sending the message to the remote partition. It is called whenever the `Asynchronous` pragma is specified for the remotely called procedure. `Establish_RPC_Receiver` is called immediately after elaborating an active partition, but prior to invoking the main subprogram, if any. The `Receiver` parameter designates an implementation-provided procedure that receives a message and calls the appropriate remote call interface package and subprogram.

6.8.4 Java

Although Java provides a convenient way of accessing network protocols, these protocols are still complex and are a deterrent to writing distributed applications. Consequently, Java also supports the distributed object communication model through the notion of **remote objects**.

Program 6.2 The Ada System.RPC package.

```

with Ada.Streams;
package System.RPC is
  type Partition_ID is range 0 .. implementation_defined;

  Communication_Error : exception;

  type Params_Stream_Type ...

  -- Synchronous call
  procedure Do_RPC(
    Partition : in Partition_ID;
    Params    : access Params_Stream_Type;
    Result    : access Params_Stream_Type);

  -- Asynchronous call
  procedure Do_APC(
    Partition : in Partition_ID;
    Params    : access Params_Stream_Type);

  -- The handler for incoming RPCs
  type RPC_Receiver is access procedure(
    Params : access Params_Stream_Type;
    Result : access Params_Stream_Type);

  procedure Establish_RPC_Receiver(Partition : Partition_ID;
    Receiver : in RPC_Receiver);
private
  ...
end System.RPC;

```

The Java model is centred on the use of the `java.rmi` package which builds on top of the TCP protocol. Within this package is the `Remote` interface:

```
public interface Remote { };
```

This is the starting point for writing distributed Java applications. Extensions of this interface are written to provide the link between the clients and servers. For example, consider again the server that wishes to return details of the local weather forecast for its location. An appropriate interface might be:

```

public interface WeatherForecast extends java.rmi.Remote {
  // shared between clients and server
  public String getForecast() throws RemoteException;
}

```

The method `getForecast` must have a `RemoteException` class in its throws list so that the underlying implementation can indicate that the remote call has failed.

Forecast is an object which has details of today's forecast. As the object will be copied across the network, it must implement the `Serializable` interface:²

```
public class Forecast implements java.io.Serializable
{
    public String Today() {
        String today = "Wet";

        return today ;
    }
}
```

Once the appropriate remote interface has been defined, a server class can be declared. Again, it is usual to indicate that objects of the class can potentially be called remotely by extending one of the predefined classes in the package `java.rmi.server`. Currently, there are two classes: `RemoteServer` (which is an abstract class derived from the `Remote` class) and `UnicastRemoteObject` which is a concrete extension of `RemoteServer`. The latter provides the class for servers which are non-replicated and have a point-to-point connection to each client using the TCP protocol. It is anticipated that future extension to Java might provide other classes such as a replicated server using a multicast communication protocol.

The following example shows a server class which provides the weather forecast for the county of Yorkshire in the UK:

```
public class YorkshireWeatherForecast
    extends UnicastRemoteObject
    implements WeatherForecast {
    public YorkshireWeatherForecast() throws RemoteException {
        super(); // call parent constructor
    }

    public Forecast getForecast() throws RemoteException {
        ...
    }
}
```

Once the server class has been written, it is necessary to generate the server and client stubs for each of the methods that can be called remotely. Note that Java uses the term **skeleton** for the server stub. The Java programming environment provides a tool called 'rmic' that will take a server class and automatically generate the appropriate client stub and server skeleton. Stubs can also be generated dynamically using the Java reflection mechanisms.

All that is now required is for the client to be able to acquire a client stub which accesses the server object. This is achieved via a registry. The registry is a separate Java program which executes on each host machine which has server objects. It listens on a standard TCP port and provides an object of the class `Naming` extracted in Program 6.3.

²Like the `Remote` interface, the `Serializable` interface is empty. In both cases it acts as a tag to give information to the compiler. For the `Serializable` interface, it indicates that the object can be converted into a stream of bytes suitable for I/O.

Program 6.3 An extract of the Java Naming class.

```

public final class Naming
{
    public static void bind(String name, Remote obj)
        throws AlreadyBoundException, java.net.MalformedURLException,
            UnknownHostException, RemoteException;
    // bind the name to the obj
    // name takes the form of a URL such as
    //      rmi://remoteHost:port/objectName

    public static Remote lookup (String name)
        throws NotBoundException, java.net.MalformedURLException,
            UnknownHostException, RemoteException;
    // looks up the name in the registry and returns a remote object

    ...
}

```

Each server object can use the Naming class to bind its remote object to a name. Clients then can access a remote registry to acquire a reference to the remote object. Of course, this is a reference to a client stub object for the server. The client stub is loaded into the client's machine. Once the reference has been obtained, the server's methods can be invoked.

6.8.5 CORBA

The Common Object Request Broker Architecture (CORBA) provides the most general distributed object model. Its goal is to facilitate interoperability between applications written in different languages, for different platforms supported by middleware software from different vendors. It was designed by the Object Management Group (OMG) – a consortium of software vendors, software developers and end-users – according to the Object Management Architectural Model, which is depicted in Figure 6.2.

At the heart of the architecture is the **Object Request Broker (ORB)**. This is a software communication bus that provides the main infrastructure that facilitates interoperability between heterogeneous applications. The term CORBA often refers to the ORB. The other components of the architecture are:

- **object services** – a collection of basic services which support the ORB; for example, support for object creation, naming and access control, and tracking relocated objects.
- **common facilities** – a set of functions which are common across a wide range of application domains; for example, user interfaces, document and database management.
- **domain interfaces** – a group of interfaces which support particular application domains such as banking and finance or telecommunications.
- **application interfaces** – the end-users' specific application interfaces.

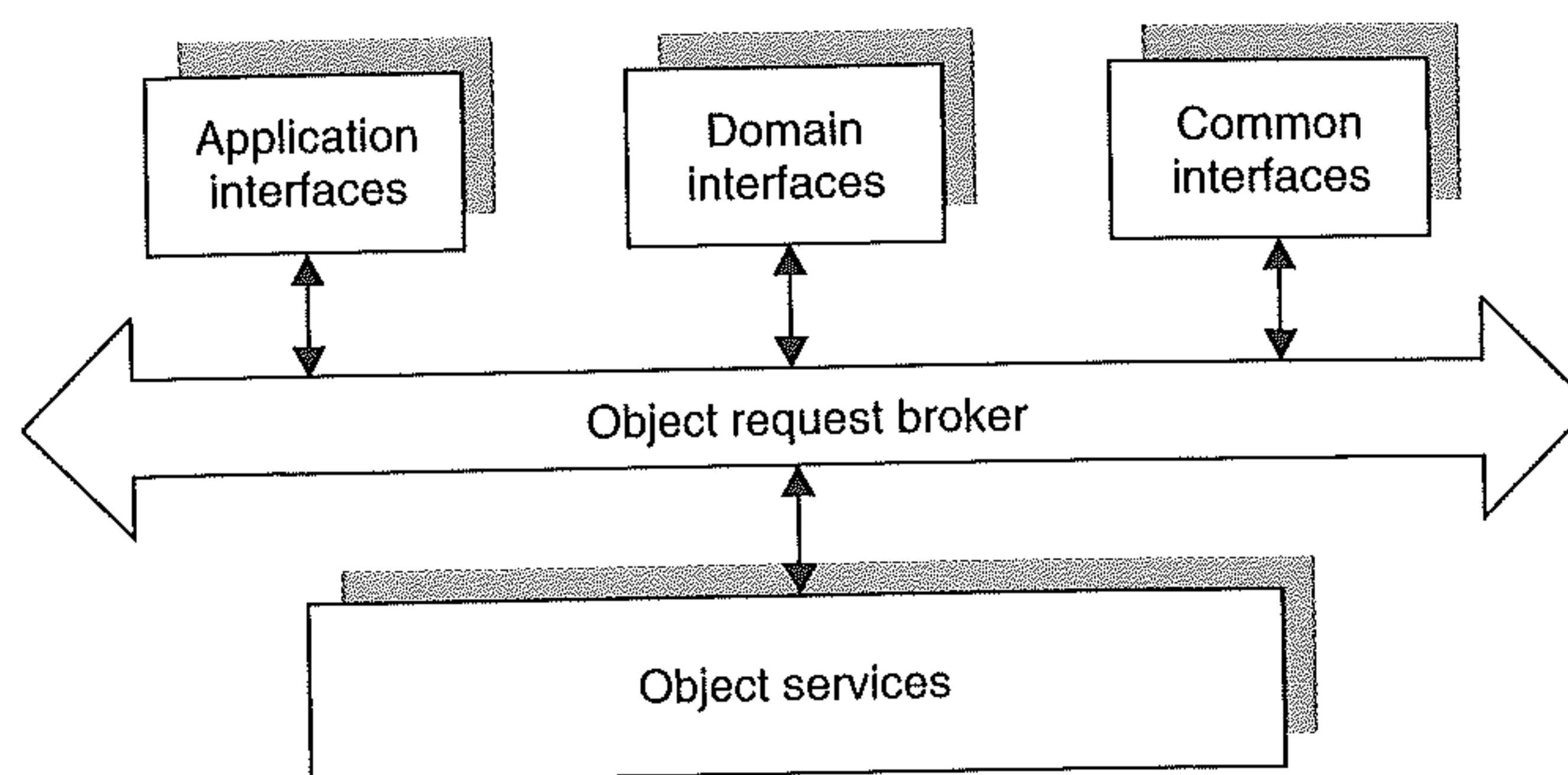


Figure 6.2 The Object Management Architecture Model.

To ensure interoperability between ORBs from different vendors, CORBA defines a General Inter-ORB Protocol which sits on top of TCP/IP.

Central to writing CORBA applications is the **Interface Definition Language (IDL)**. A CORBA interface is similar in concept to the Java remote interface discussed in the previous section. The IDL (which is like the C++ language) is used to describe the facilities to be provided by an application object, the parameters to be passed with a given method and its return values, along with any object attributes. An example interface for the weather forecast application given in the previous section is shown below:

```

interface WeatherForecast {
    void GetForecast(out Forecast today);
}
  
```

Once the IDL for the application is defined, tools are used to ‘compile’ it. The IDL compiler generates several new files in one of a number of existing programming languages such as C, Java or Ada. The files include:

- client stubs that provide a communication channel between the client and the ORB; and
- a server skeleton which enables the ORB to call functions on the server.

The code for the server and clients can now be written. The server consists of two parts: the code for the application object itself and the code for the server process. The application object has to be associated with the server skeleton. The way this is achieved is dependent on the target language. For example in Java, it can be done by producing a class which is a subclass of the generated server skeleton. The methods for the application objects are then completed. The server process itself can be written as a main program which creates the application object and initializes the ORB and tells it that the object is ready to receive client requests. The structure of the client is similar.

In reality, CORBA provides many more facilities than those described above. As well as the static association between client and server, CORBA clients can dynamically discover a server’s IDL interface without prior knowledge of the server details. Furthermore, the services that support the ORB allow for a wide range of functionality such

as an event service, transaction and concurrency control, persistent objects and trading services. Applications access these services via the Portable Object Adapter (POA). This is a library which provides the run-time environment for a server object.

More recently, the CORBA Component Model (CCM) (Object Management Group, 2002) has been introduced. The CCM extends the CORBA object model to support the concept of components and establishes standards for implementing, packaging, assembling and deploying component implementations. Components are larger building blocks than objects. From a client perspective, a CCM component is an extended CORBA object that encapsulates various interaction models via different interfaces and connection operations. From a server perspective, components are units of implementation that can be installed and instantiated independently in standard application server run-time environments. Remote method invocation is still the primary communication mechanism.

6.9 Simple embedded system revisited

In Section 4.8, a simple embedded system was introduced and a concurrent solution was proposed. Then in Section 5.11 the Ada solution was updated to illustrate communication with the operator console via Ada's protected objects mechanism. Here the example is repeated using the Ada rendezvous.

The body of the I/O routines can now be completed using a rendezvous. The data to be sent to the console is communicated synchronously via entry calls to the console task. The console task waits using the `select` statement.

```
package body IO is
  task Console is
    entry Write(R : Temp_Reading);
    entry Write(R : Pressure_Reading);
  end Console;

  -- procedures for data exchange with the environment
  procedure Read(TR : out Temp_Reading) is separate; -- from DAC
  procedure Read(PR : out Pressure_Reading) is separate; -- from DAC
  procedure Write(HS : Heater_Setting) is separate; -- to switch.
  procedure Write(PS : Pressure_Setting) is separate; -- to DAC

  task body Console is
    Last_Temperature : Temp_Reading;
    Last_Pressure : Pressure_Reading;
  begin
    ...
    loop
      select
        accept Write(R : Temp_Reading) do
          Last_Temperature := R;
        end Write;
        -- output reading to console
      or
        accept Write(R : Pressure_Reading) do
          Last_Pressure := R;
```



```

        end Write;
        -- output reading to console
    or
        terminate;
    end select;
    ...
end loop;
end Console;

procedure Write(TR : Temp_Reading) is
begin
    Console.Write(TR);
end Write; -- to screen

procedure Write(PR : Pressure_Reading) is
begin
    Console.Write(PR);
end Write; -- to screen
end IO;

```

Summary

Ada and C/Real-Time POSIX provide a message-passing facility; they also support alternative communication paradigms.

The semantics of message-based communication are defined by three issues:

- the model of synchronization;
- the method of task naming;
- the message structure.

Variations in the task synchronization model arise from the semantics of the send operation. Three broad classifications exist:

- **asynchronous** – sender task does not wait;
- **synchronous** – sender task waits for message to be read;
- **remote invocation** – sender task waits for message to be read, acted upon and a reply generated.

Remote invocation can be made to appear syntactically similar to a procedure call. This can cause confusion when remote procedure calls (RPCs) are used in a distributed system. RPCs are, however, an implementation strategy; remote invocation defines the semantics of a particular message-passing model. The two tasks involved in this communication may be on the same processor, or they may be distributed; the semantics are the same.

Process naming involves two distinct issues; direct or indirect, and symmetry. Ada uses remote invocation with direct asymmetric naming. Messages can take the form of any system or user-defined type. C/Real-Time POSIX supports an asynchronous symmetrical scheme.

Communication in Ada requires one task to define an entry and then, within its body, accept any incoming call. A rendezvous occurs when one task calls an entry in another and it is accepted. In C/Real-Time POSIX, communication is via message queues with send/receive primitives. Ada supports a many-to-one communication mechanism, whereas C/Real-Time POSIX supports a many-to-many scheme.

In order to increase the expressive power of message-based concurrent programming languages, it is necessary to allow a task to choose between alternative communications. The language primitive that supports this facility is known as *selective waiting*. Here, a task can choose between different alternatives; on any particular execution some of these alternatives can be closed off by using a boolean guard. Ada provides the `select` statement to support this facility. It has two extra facilities.

- (1) A `select` statement may have an `else` part which is executed if there are no outstanding calls on open alternatives.
- (2) A `select` statement may have a terminate alternative which will cause the task executing the select to terminate if none of the other tasks that could call it are still executable.

In C/Real-Time POSIX, a process or thread can indicate that it is not prepared to block when the message queue is full or empty. A notification mechanism is used to allow the operating system to send a signal when a process can continue. This mechanism can be used to wait for a message on one or more message queues.

An important feature of Ada's extended rendezvous is its interaction with the exception-handling model. If an exception is raised but not handled during a rendezvous then it is propagated to both the calling and called tasks. A calling task must, therefore, protect itself against possibly anonymous exceptions being generated as a result of making a message-passing call.

For distributed systems, Ada allows a collection of library units to be grouped into 'partitions' that communicate via remote procedure calls and remote object invocations. Neither configuration, nor allocation nor reconfiguration are supported directly by the language.

Java allows objects to be distributed which can communicate via remote procedure calls or via sockets. Again configuration, allocation or reconfiguration are not supported directly by the language.

CORBA facilitates distributed objects and interoperability between applications written in different languages for different platforms supported by middleware software from different vendors. An extensive set of capabilities is provided.

Further reading

- Andrews, G. A. (1991) *Concurrent Programming Principles and Practice*. Redwood City, CA: Benjamin/Cummings.
- Burns, A. and Davies, G. (1993) *Concurrent Programming*. Reading, MA: Addison-Wesley.

- Burns, A. and Wellings, A. J. (2007) *Concurrent and Real-Time Programming in Ada*. Cambridge: Cambridge University Press.
- Hoare, C. A. R. (1985) *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice Hall.
- Siegel, J. (2001) *CORBA 3 Fundamentals and Programming*, 2nd edn. New York: Wiley.
- Silberschatz, A., Galvin P. A. and Gagne, G. (2008) *Operating System Concepts*, 8th edn. New York: Wiley.

Exercises

- 6.1 Given that Ada supports protected objects, should the Ada rendezvous facilities be removed from the language?
- 6.2 If an Ada task has two entry points, is it possible for it to accept the entry of the calling task which has been waiting for the longest period of time?
- 6.3 Show how to implement a binary semaphore using an Ada task and a rendezvous. What happens if a task issuing a `Wait` is aborted before it can issue a `Signal`?
- 6.4 Discuss the advantages and disadvantages of implementing semaphores with a rendezvous rather than a protected object.
- 6.5 Show how an Ada task and the rendezvous can be used to implement Hoare's monitors.
- 6.6 To what extent can an Ada rendezvous be considered a Remote Procedure Call? Discuss the following code which purports to implement a particular remote procedure. Consider both the called procedure and the implications for the calling task.

```

task type Rpc_Implementation is
  entry Rpc (Param1:Type1; Param2:Type2);
end Rpc_Implementation;

task body Rpc_Implementation is
begin
  accept Rpc (Param1:Type1; Param2:Type2) do
    -- body of procedure
  end Rpc;
end Rpc_Implementation;

-- declare an array of 1000 rpc_implementation tasks
Concurrent_Rpc : array(1 .. 1000) of Rpc_Implementation;

```

- 6.7 Complete Exercise 4.8 using the Ada tasks and the rendezvous.
- 6.8 Consider an Ada system of three cigarette smoker tasks and one agent task. Each smoker continually makes cigarettes and smokes them. To make a cigarette, three ingredients are required: tobacco, paper and matches. One of the smoker tasks has an infinite supply of paper, another has an infinite supply of tobacco and the third has an infinite supply of matches. The agent task has an infinite supply of all three ingredients. Each smoker task must communicate with the agent task to obtain the two ingredients it is missing.

The agent task has the following specification:

```
task Agent is
  entry Give_Matches(...);
  entry Give_Paper(...);
  entry Give_Tobacco(...);
  entry Cigarette_Finished;
end Agent;
```

The body of the agent task chooses two ingredients randomly, and then accepts communication on the associated entries to pass the ingredients to the smokers. Once both ingredients have been passed, it waits for communication on the `Cigarette_Finished` entry before repeating the task indefinitely. A smoker, having received the required ingredients, makes and smokes a cigarette, indicates completion by calling the `Cigarette_Finished` entry and then requests new ingredients.

Sketch the specification and body of the three smoker tasks and the body of the agent task. If necessary, add parameters to the entry specifications of the Agent task. The solution should be deadlock-free.

6.9 A server task has the following Ada specification:

```
task Server is
  entry Service_A;
  entry Service_B;
  entry Service_C;
end Server;
```

Write the body of the Server task so that it performs all of the following operations.

- If client tasks are waiting on all the entries, the task should service the clients in a cyclic order; that is, accept first a `Service_A` entry, and then a `Service_B` entry, and then a `Service_C` entry, and then a `Service_A` entry and so on.
- If not all entries have a client task waiting, the Server should service the other entries in a cyclic order. The Server tasks should not be blocked if there are clients still waiting for a service.
- If the Server task has no waiting clients then it should NOT busy-wait; it should block waiting for a client's request to be made.
- If all the possible clients have terminated, the Server should terminate.

Assume that client tasks are not aborted and issue simple entry calls only.

6.10 The following Ada package provides a service. During the provision of this service the exceptions A, B, C and D can be raised.

```
package Server is
  A, B, C, D : exception;
  procedure Service; -- can raise A, B, C or D
end Server;
```

In the following procedure two tasks are created; task One rendezvous with task Two. During the rendezvous task, Two calls the Service provided by the server package.

```

with Server; use Server;
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
  task One;

  task Two is
    entry Sync;
  end Two;

  task body One is
  begin
    Two.Sync;
  exception
    when A =>
      Put_Line("A trapped in one ");
      raise;
    when B =>
      Put_Line("B trapped in one");
      raise C;
    when C =>
      Put_Line("C trapped in one");
    when D =>
      Put_Line("D trapped in one");
  end;

  task body Two is
  begin -- block X
    begin -- block Y
      begin -- block Z
        accept Sync do
          begin
            Service;
          exception
            when A =>
              Put_Line("A trapped in sync");
            when B =>
              Put_Line("B trapped in sync");
              raise;
            when C =>
              Put_Line("C trapped in sync");
              raise D;
          end;
        end Sync;
      exception
        when A =>
          Put_Line("A trapped in block Z");
        when B =>
          Put_Line("B trapped in block Z");
          raise C;
        when others =>

```



```

        Put_Line("others trapped in Z");
        raise C;
    end; -- block Z
exception
    when C =>
        Put_Line("C trapped in Y");
    when others =>
        Put_Line("others trapped in Y");
        raise C;
    end; -- block Y
exception
    when A =>
        Put_Line("A trapped in X");
    when others =>
        Put_Line("others trapped in X");
end; -- block X and task TWO

begin    -- procedure main
    null;

exception
    when A =>
        Put_Line("A trapped in main");
    when B =>
        Put_Line("B trapped in main");
    when C =>
        Put_Line("C trapped in main");
    when D =>
        Put_Line("D trapped in main");

end Main;

```

The procedure `Put_Line` is declared in the package `Text_IO`, and when called it prints its argument on the terminal.

What output would appear if the `Service` procedure:

- (1) raised exception A?
- (2) raised exception B?
- (3) raised exception C?
- (4) raised exception D?

Assume that output does not become intermingled by concurrent calls to `Put_Line`.

Chapter 7

Atomic actions, concurrent tasks and reliability

7.1	Atomic actions	7.6	Asynchronous notification in Ada
7.2	Atomic actions in C/Real-Time POSIX, Ada and Real-Time Java	7.7	Asynchronous notification in Real-Time Java
7.3	Recoverable atomic actions		Summary
7.4	Asynchronous notification		Further reading
7.5	Asynchronous notification in C/Real-Time POSIX		Exercises

Chapter 2 considered how reliable software could be produced in the presence of a variety of errors. Modular decomposition and atomic actions were identified as two techniques essential for damage confinement and assessment. Also, the notions of forward and backward error recovery were introduced as approaches to dynamic error recovery. It was shown that where tasks communicate and synchronize their activities, backward error recovery may lead to the domino effect. In Chapter 3, exception handling was discussed as a mechanism for providing both forward and backward error recovery in sequential tasks. Chapters 4, 5 and 6 then considered the facilities provided by operating systems and real-time languages for concurrent programming. This chapter brings together exception handling and concurrency in order to show how tasks can interact reliably in the presence of other tasks and in the presence of faults. The notion of an atomic action is explored in more detail and the concept of asynchronous notification is introduced.

In Chapter 4, the interaction of tasks was described in terms of three types of behaviour:

- independent
- cooperation
- competition.

Independent tasks do not communicate or synchronize with each other. Consequently, if an error occurs within one task, then recovery procedures can be initiated by that task in isolation from the rest of the system. Recovery blocks and exception handling can be used as described in Chapters 2 and 3.

Cooperating tasks, by comparison, regularly communicate and synchronize their activities in order to perform some common operation. If any error condition occurs, it is necessary for all tasks involved to perform error recovery. The programming of such error recovery is the topic of this chapter.

Competing tasks communicate and synchronize in order to obtain resources; they are, however, essentially, independent. An error in one should have no effect on the others. Unfortunately, this is not always the case, particularly if the error occurred while a task was in the act of being allocated a resource. Reliable resource allocation is considered in Chapter 8.

Where cooperating tasks communicate and synchronize through shared resources, recovery may involve the resource itself. This aspect of resource allocation will also be considered in Chapter 8.

7.1 Atomic actions

One of the main motivations for introducing concurrent tasks into a language is that they enable parallelism in the real world to be reflected in application programs. This enables such programs to be expressed in a more natural way and leads to the production of more reliable and maintainable systems. Disappointingly, however, concurrent tasks create many new problems which did not exist in the purely sequential program. Consequently, the last few chapters have been dedicated to discussing some of the solutions to these problems: in particular, communication and synchronization between tasks using shared variables (correctly) and message passing. This was undertaken in a fairly isolated manner and no consideration has yet been given to the way in which groups of concurrent tasks should be structured in order to coordinate their activities.

The interaction between two tasks has, so far, been expressed in terms of a single communication. In reality, this is not always the case. For example, withdrawal from a bank account may involve a ledger task and a payment task in a sequence of communications to authenticate the drawer, check the balance and pay the money. Furthermore, it may be necessary for more than two tasks to interact in this way to perform the required action. In all such situations, it is imperative that the tasks involved see a consistent system state. With concurrent tasks, it is all too easy for groups of tasks to interfere with one other.

What is required is for each group of tasks to execute their joint activity as an **indivisible** or **atomic action**. Of course, a single task may also want to protect itself from the interference of other tasks (for example, during resource allocation). It follows that an atomic action may involve one or more tasks. Atomic actions have also been called *multiparty* interactions (Evangelist et al., 1989; Yuh-Jzer and Smolka, 1996).

There are several almost equivalent ways of expressing the properties of an atomic action (Lomet, 1977; Randell et al., 1978).

- (1) An action is atomic if the tasks performing it are not aware of the existence of any other active task, and no other active task is aware of the activity of the tasks during the time the tasks are performing the action.
- (2) An action is atomic if the tasks performing it do not communicate with other tasks while the action is being performed.

- (3) An action is atomic if the tasks performing it can detect no state change except those performed by themselves and if they do not reveal their state changes until the action is complete.
- (4) Actions are atomic if they can be considered, so far as other tasks are concerned, to be indivisible and instantaneous, such that the effects on the system are as if they were interleaved as opposed to concurrent.

These are not quite all equivalent. For example, consider the second expression: an action is atomic if the tasks performing it communicate only among themselves and not with other tasks in the system. Unlike the other three, this does not really define the true nature of an atomic action. While it will guarantee that the action is indivisible, it is too strong a constraint on the tasks. Interactions between an atomic action and the rest of the system can be allowed as long as they have no impact on the activity of the atomic action and do not provide the rest of the system with any information concerning the progress of the action (Anderson and Lee, 1990). In general, in order to allow such interactions requires detailed knowledge of the atomic action's function and its interface to the rest of the system. As this cannot be supported by a general language implementation, it is tempting, following Anderson and Lee (1990), to adopt the more restrictive (second) definition. This can only be done, however, if the resources necessary to complete an atomic action are acquired by the underlying implementation, not by instructions given in the program. If resources are to be acquired and released when the programmer desires, tasks within atomic actions will have to communicate with general-purpose resource managers.

Although an atomic action is viewed as being indivisible, it can have an internal structure. To allow modular decomposition of atomic actions, the notion of a **nested atomic action** is introduced. The tasks involved in a nested action must be a subset of those involved in the outer level of the action. If this were not the case, a nested action could smuggle information concerning the outer level action to an external task. The outer level action would then no longer be indivisible.

7.1.1 Two-phase atomic actions

Ideally, all tasks involved in an atomic action should obtain the resources they require (for the duration of the action) prior to its commencement. These resources could then be released after the atomic action had terminated. If these rules were followed, there would be no need for an atomic action to interact with any external entity and the stricter definition of atomic action could be adopted.

Unfortunately, this ideal can lead to poor resource utilization, and hence a more pragmatic approach is needed. Firstly, it is necessary to allow an atomic action to start without its full complement of resources. At some point, a task within the action will request a resource allocation; the atomic action must then communicate with the resource manager. This manager may be a server task. If a strict definition of atomic action is adhered to, this server would have to form part of the atomic action, with the effect of serializing all actions involving the server. Clearly, this is undesirable, and hence an atomic action is allowed to communicate externally with resource servers.

Within this context, a resource server is defined to be a custodian of non-sharable system utilities. It protects these utilities against inappropriate access, but does not, itself, perform any actions upon them.

A further improvement in resource allocation can be made if a task is allowed to release a resource prior to completion of the associated atomic action. In order for this premature release to make sense, the state of the resource must be identical to that which would appertain if the resource was retained until completion of the atomic action. Its early release will, however, enhance the concurrency of the whole system.

If resources are to be obtained late and released early it could be possible for an external state change to be affected by a released resource and observed by the acquisition of a new resource. This would break the definition of atomic action. It follows that the only safe policy for resource usage is one that has two distinct phases. In the first 'growing' phase, resources can be requested (only); in the following 'shrinking' phase, resources can be released (but no new allocations can be made). With such a structure, the integrity of the atomic action is assured. However, it should be noted that if resources are released early then it will be more difficult to provide recovery if the atomic action fails. This is because the resource has been updated and another task may have observed the new state of the resource. Any attempt to invoke recovery in the other task may lead to the domino effect (see Section 2.5.3).

In all the following discussions, atomic actions are assumed to be two-phased; recoverable actions do not release any resources until the action successfully completes.

7.1.2 Atomic transactions

Within the theories of operating systems and databases, the term **atomic transaction** is often used. An atomic transaction has all the properties of an atomic action plus the added feature that its execution is allowed either to succeed or to fail. By failure, it is meant that an error has occurred from which the transaction cannot recover; for example, a processor failure. If an atomic action fails then the components of the system, which are being manipulated by the action, may be left in an inconsistent state. With an atomic transaction, this cannot happen because the components are returned to their original state (that is, the state they were *before* the transaction commenced). Atomic transactions are sometimes called **recoverable actions** and, unfortunately, the terms **atomic action** and **atomic transaction** are often interchanged.

The two distinctive properties of atomic transactions are:

- **failure atomicity** – meaning that the transaction must either complete successfully or (in the case of failure) have no effect;
- **synchronization atomicity** (or isolation) – meaning that the transaction is indivisible in the sense that its partial execution cannot be observed by any concurrently executing transaction.

Although atomic transactions are useful for those applications which involve the manipulation of databases, they are not suitable for programming fault-tolerant systems *per se*. This is because they imply that some form of recovery mechanism will be supplied by the system. Such a mechanism would be fixed, with the programmer having no control over

its operation. Although atomic transactions provide a form of backward error recovery, they do not allow recovery procedures to be performed. Notwithstanding these points, atomic transactions do have a role in protecting the integrity of a real-time database system.

7.1.3 Requirements for atomic actions

If a real-time programming language is to be capable of supporting atomic actions, it must be possible to express the requirements necessary for their implementation. These requirements are independent of the notion of a task and the form of intertask communication provided by a language. They are the following.

- **Well-defined boundaries** – each atomic action should have a start, an end and a side boundary. The start boundary is the location in each task involved in the atomic action where the action is deemed to start. The end boundary is the location in each task involved in the atomic action where the action is deemed to end. The side boundary separates those tasks involved in the atomic action from those in the rest of the system.
- **Indivisibility (isolation)** – an atomic action must not allow the exchange of any information between the tasks active inside the action and those outside (resource managers excluded). If two atomic actions do share data then the value of that data after the atomic actions is determined by the strict sequencing of the two actions in some order.

There is no implied synchronization at the start of an atomic action. Tasks can enter at different times. However, there is an implied synchronization at the end of an atomic action; tasks are not allowed to leave the atomic action until all tasks are willing and able to leave.

- **Nesting** – atomic actions may be nested as long as they do not overlap with other atomic actions. Consequently, in general, only strict nesting is allowed (two structures are strictly nested if one is completely contained within the other).
- **Concurrency** – it should be possible to execute different atomic actions concurrently. One way to enforce indivisibility is to run atomic actions sequentially. However, this could seriously impair the performance of the overall system and therefore should be avoided. Nevertheless, the overall effect of running a collection of atomic actions concurrently must be the same as that which would be obtained from serializing their executions.
- As it is the intention that atomic actions should form the basis of damage confinement, they must allow recovery procedures to be programmed.

Figure 7.1 diagrammatically represents the boundaries of a nested atomic action in a system of six tasks. Action B involves only tasks P_3 and P_4 , whereas action A also includes P_2 and P_5 . The other tasks (P_1 and P_6) are outside the boundaries of both atomic actions.

It, perhaps, should be noted at this point that some definitions of atomic actions require that all tasks be synchronized on *both* entry and exit of the action.

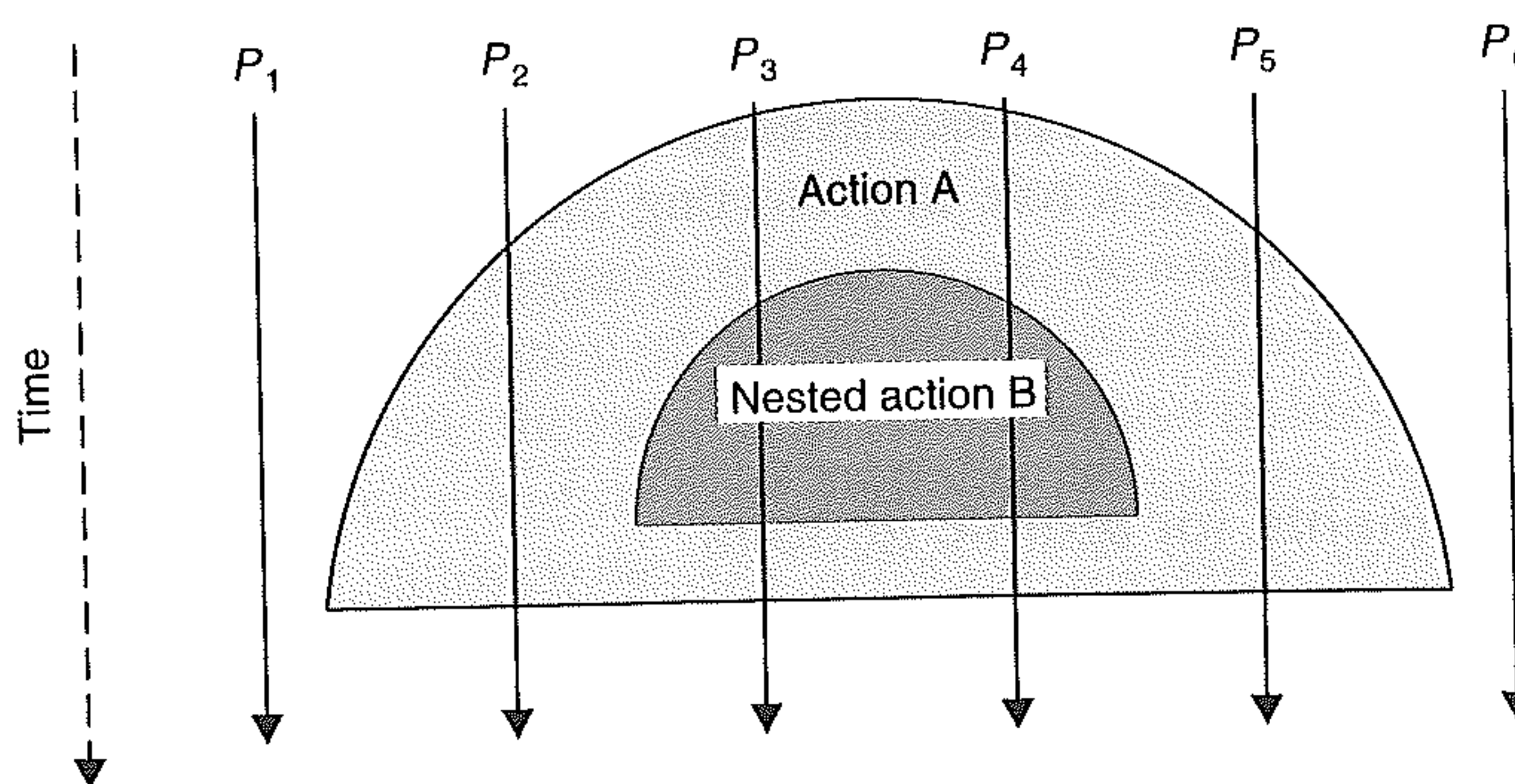


Figure 7.1 Nested atomic actions.

7.2 Atomic actions in C/Real-Time POSIX, Ada and Real-Time Java

Atomic actions provide structuring support for the software of large embedded systems. To get the full benefit of this aid requires the support of the real-time language. Unfortunately, such support is not directly provided by any of the major languages. This section considers the suitability of C/Real-Time POSIX, Ada and Real-Time Java for programming atomic actions. Following this, a possible language framework is given, and then this framework is extended to provide forward and backward error recovery.

The problem of resource allocation is postponed until Chapter 8. For now, it is assumed that resources have two modes of use: sharable and non-sharable, with some resources being amenable to both sharable and non-sharable modes. Furthermore, it is assumed that all actions are two-phased, and that the resource manager will ensure that appropriate usage is made of the resources. Also tasks within an action synchronize their own access to the resource to avoid any interference.

Atomic actions could be encapsulated within the monitor-like construct that is found in each of the languages. However, this would not allow any parallelism within an action. Hence, an alternative approach is adopted. In a similar fashion to the solution to the reader/writers problem presented in Section 5.9.1, the entry and exit protocols are implemented with a monitor-like construct and then the application code is surrounded by calls to the routines. Figure 7.2 illustrates an atomic action that requires three tasks (T_1 , T_2 and T_3). Each task has its own role to play in the action. These roles are represented by the action procedures: for example, T_1 calls Action Procedure 1, etc. These action procedures then liaise with an action controller to provide the required synchronization. Figure 7.3 shows the structure of the roles.

7.2.1 Atomic actions and C/Real-Time POSIX mutexes

Using C and the Real-Time POSIX API, support for atomic actions following the structure given above can be provided as follows. Here only a two thread action is illustrated, but the approach is easily extended to more threads.

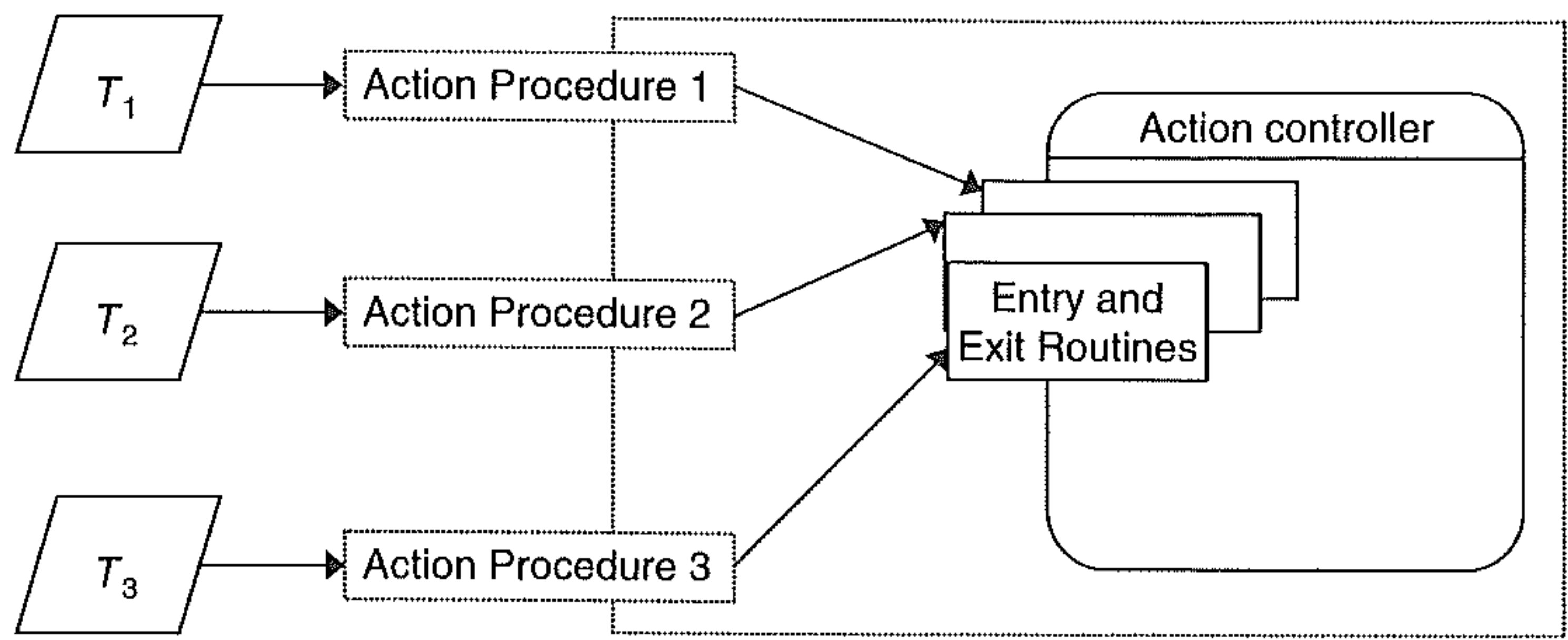


Figure 7.2 The structure of an action controller.

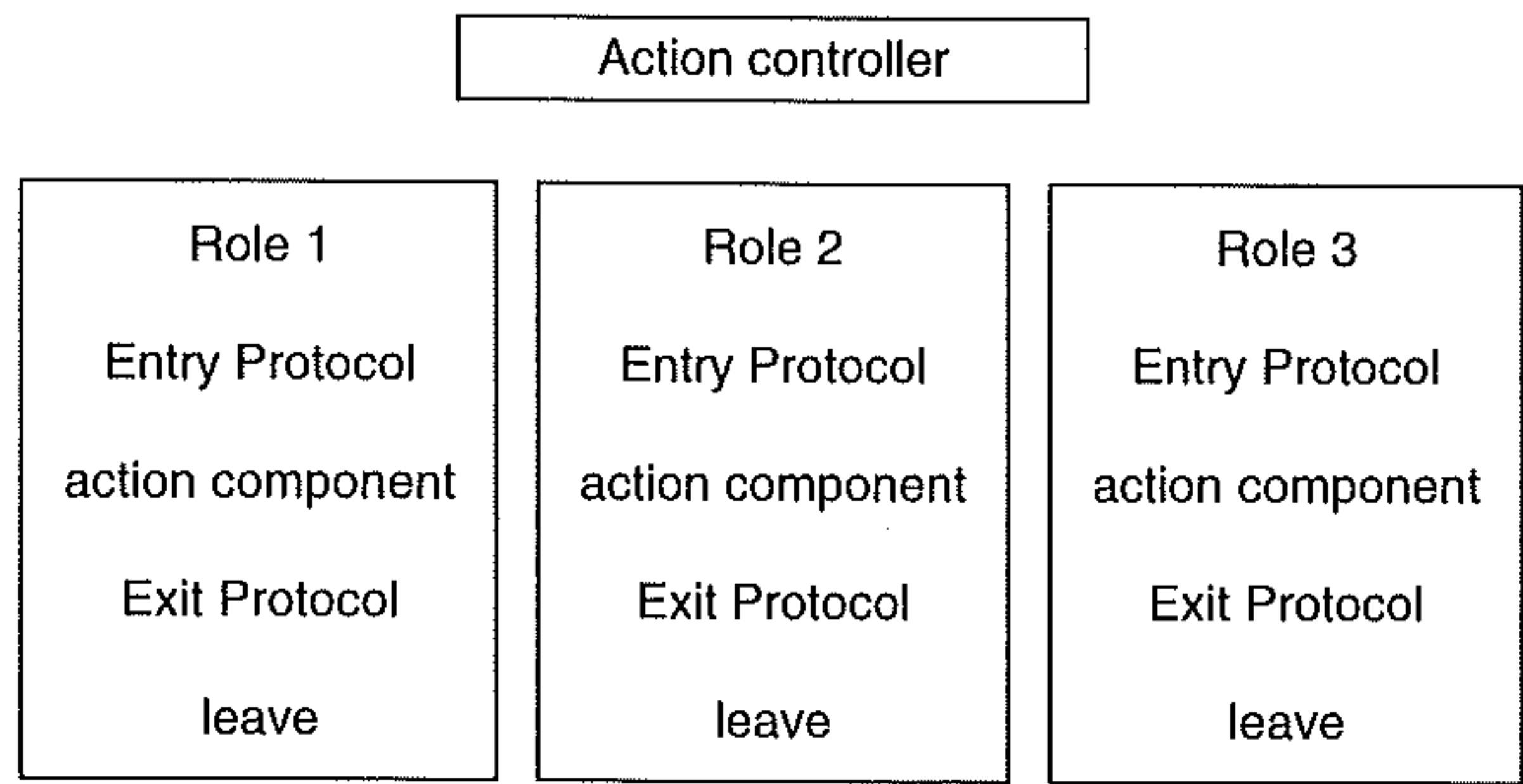


Figure 7.3 Using the action controller.

First, the mutex, condition variables and shared variables needed by the action controller are encapsulated in a `typedef`. Several boolean variables are used to represent the state of the action, and a mutex and four condition variables (and their associated attribute objects) provide the necessary synchronization.

```

#include <pthread.h>

typedef unsigned int boolean;
#define false 0
#define true (!false)

typedef struct {
    boolean first_process_active;
    boolean second_process_active;
    boolean first_process_finished;
    boolean second_process_finished;

    pthread_mutexattr_t mutex_att;
    pthread_mutex_t mutex;

    pthread_condattr_t cond_att;
    pthread_cond_t no_first_process, no_second_process;
    pthread_cond_t atomic_action_ends1, atomic_action_ends2;
} monitor;
```

The entry and exit protocols in support of the first role can now be given:

```
void entry_1(monitor *M) {
    PTHREAD_MUTEX_LOCK(&M->mutex);
    while(M->first_process_active)
        PTHREAD_COND_WAIT(&M->no_first_process, &M->mutex);
    M->first_process_active = true;
}

void exit_1(monitor *M) {
    while(!M->second_process_finished)
        PTHREAD_COND_WAIT(&M->atomic_action_ends2, &M->mutex);
    PTHREAD_COND_SIGNAL(&M->atomic_action_ends1);
    M->first_process_active = false;
    PTHREAD_COND_SIGNAL(&M->no_first_process);
    PTHREAD_MUTEX_UNLOCK(&M->mutex);
}
```

To enter the action, the action controller first obtains a mutual exclusion lock so it knows that its action will not be interfered with. It then checks that there is no other thread executing this role. If there is, the controller forces the calling thread to wait until the role is available. If the role is available, the thread is allowed to continue and a note is made that the role is now active.

When the role is finished, the exit protocol is called. Here, the controller checks to see if the other thread has finished its role. If it hasn't the first thread is made to wait. If it has, the action controller signals that the first thread has finished its role, then returns the data structure to a position from which a new action can start. The second role is similarly structured.

On exit from its role, the thread signals that it has finished and then waits for the other thread to finish. Of course, the data structure must be initialized via some appropriate routine:

```
void init_action(monitor *M) {
    // initialize all mutex and conditions with the appropriate
    // attributes; initialize booleans to false
}
```

The structure of the application code takes the following form:

```
void code_for_first_process(monitor *M) {
    entry_1(M);
    // get resource in non-sharable mode
    // update resource

    // signal second process that it is ok
    // for it to access resource

    // any final processing
    exit_1(M);
}
```



```

void code_for_second_process (monitor *M) {
    entry_2(M);
    // initial processing

    // wait for first process to signal
    // that it is ok to access resource

    // access resource

    // release resource
    exit_2(M);
}

```

Although the above approach has been able to implement the synchronization aspects of an atomic action, there is no way to ensure that the application threads do not communicate with any other threads.

7.2.2 Atomic actions in Ada

The Ada approach to implementing an atomic action is illustrated below for a three task system. It is similar to the C/Real-Time POSIX approach but uses guards instead of condition variables. Here, the action is encapsulated within an Ada package: ,

```

package Action_X is
    procedure Code_For_First_Task(--params);
    procedure Code_For_Second_Task(--params);
    procedure Code_For_Third_Task(--params);
end Action_X;

```

The body of the package contains the action controller, which is implemented as a protected type.

```

package body Action_X is
    protected Action_Controller is
        entry First;
        entry Second;
        entry Third;
        entry Finished;
    private
        First_Here : Boolean := False;
        Second_Here : Boolean := False;
        Third_Here : Boolean := False;
        Release : Boolean := False;
    end Action_Controller;

    protected body Action_Controller is
        entry First when not First_Here is
            begin
                First_Here := True;
            end First;

```

```

entry Second when not Second_Here is
begin
  Second_Here := True;
end Second;

entry Third when not Third_Here is
begin
  Third_Here := True;
end Third;

entry Finished when Release or Finished'Count = 3 is
begin
  if Finished'Count = 0 then
    Release := False;
    First_Here := False;
    Second_Here := False;
    Third_Here := False;
  else
    Release := True;
  end if;
end Finished;
end Action_Controller;

```

In the above code, the action is synchronized by the `Action_Controller` protected object. This ensures that only three tasks can be active in the action at any one time and that they are synchronized on exit. The boolean `Release` is used to program the required release conditions on `Finished`. The first two calls on `Finished` will be blocked as both parts of the barrier expression are false. When the third call comes, the `Count` attribute will become three; the barrier becomes open and one task will execute the entry body. The `Release` variable ensures that the other two tasks are both released. The last task to exit must ensure that the barrier is closed again.

The application code also has a similar structure to the C/Real-Time POSIX approach.

```

procedure Code_For_First_Task(--params) is
begin
  Action_Controller.First;
  -- acquire resources
  -- the action itself, communicates with tasks executing
  -- inside the action via resources
  Action_Controller.Finished;
  -- release resources
end Code_For_First_Task;

-- similar for second and third task
begin
  -- any initialization of local resources
end Action_X;

```

More details on how to program atomic actions in Ada can be found in Wellings and Burns (1997).

7.2.3 Atomic actions in Java

The previous sections have illustrated the basic structure for programming atomic actions. The Java approach could follow a similar structure. However, this section takes the opportunity to expand on this approach so that the Java support can easily be extended using inheritance.

First, an interface can be defined for a three-way atomic action:

```
public interface ThreeWayAtomicAction {
    public void role1();
    public void role2();
    public void role3();
}
```

Using this interface, it is possible to provide several action controllers that implement a variety of models. Applications can then choose the appropriate controller without having to change their code.

The following action controller implements the same semantics as that given previously for C/Real-Time POSIX and Ada. A synchronized `Controller` class implements the required entry and exit synchronization protocols. The `finished` method is, however, a little more complex than its Ada counterpart. This is due to the semantics of `wait` and `notifyAll`. In particular, it is necessary to count the tasks as they leave the action (using `toExit`) in order to know when to reset the internal data structures for the next action. In Ada this was achieved via the `Count` attribute.

```
public class AtomicActionControl
    implements ThreeWayAtomicAction {
    protected Controller Control;
    public AtomicActionControl() { // constructor
        Control = new Controller();
    }

    class Controller {
        protected boolean firstHere, secondHere, thirdHere;
        protected int allDone;
        protected int toExit;
        protected int numberOfParticipants;

        Controller() {
            firstHere = false;
            secondHere = false;
            thirdHere = false;
            allDone = 0;
            numberOfParticipants = 3;
            toExit = numberOfParticipants;
        }

        synchronized void first() throws InterruptedException {
            while(!firstHere) wait();
            firstHere = true;
        }
    }
}
```

```

synchronized void second() throws InterruptedException {
    while(secondHere) wait();
    secondHere = true;
}

synchronized void third() throws InterruptedException {
    while(thirdHere) wait();
    thirdHere = true;
}

synchronized void finished() throws InterruptedException {
    allDone++;
    if(allDone == numberOfParticipants) {
        notifyAll();
    } else while(allDone != numberOfParticipants) {
        wait();
    };
    toExit--;
    if(toExit == 0) {
        firstHere = false;
        secondHere = false;
        thirdHere = false;
        allDone = 0;
        toExit = numberOfParticipants;
        notifyAll(); // release all waiting for the next action
    }
}

public void role1() {
    boolean done = false;
    while(!done) {
        try {
            Control.first();
            done = true;
        } catch (InterruptedException e) {
            // ignore
        }
    }

    // .... perform action

    done = false;
    while(!done) {
        try {
            Control.finished();
            done = true;
        } catch (InterruptedException e) {
            // ignore
        }
    }
}

public void role2() {
    // similar to role1
}

```



```

public void role3() {
    // similar to role1
}

```

Given the above framework, it is now possible to extend this to produce, say, a four-way action:

```

public interface FourWayAtomicAction
    extends ThreeWayAtomicAction {
    public void role4();
}

```

and then

```

public class NewAtomicActionControl extends AtomicActionControl
    implements FourWayAtomicAction {
    public NewAtomicActionControl() {
        Control = new RevisedController();
    }

    class RevisedController extends Controller {
        protected boolean fourthHere;

        RevisedController() {
            super();
            fourthHere = false;
            numberOfParticipants = 4;
            toExit = numberOfParticipants;
        }

        synchronized void fourth() throws InterruptedException {
            while(fourthHere) wait();
            fourthHere = true;
        }

        synchronized void finished() throws InterruptedException {
            super.finished();
            if(allDone == 0) {
                fourthHere = false;
                notifyAll();
            }
        }
    }

    public void role4() {
        boolean done = false;
        while(!done) {
            try {
                // As Control is of type Controller, it must first
                // be converted to a RevisedController in order
                // to call the fourth method
                ((RevisedController)Control).fourth();
                done = true;
            } catch (InterruptedException e) {

```

```

        // ignore
    }
}

// .... perform action

done = false;
while(!done) {
    try {
        Control.finished();
        done = true;
    } catch (InterruptedException e) {
        // ignore
    }
}
}
}

```

Note that it has been necessary to override the `finished` method. Care must be taken here because of Java's automatic run-time dispatching of method calls. All calls to `finished` in the original code will dispatch to the overridden method.

7.3 Recoverable atomic actions

Although the various models described above have enabled a simple atomic action to be expressed, they all rely on programmer discipline to ensure that no interactions with external tasks occur (apart from with resource allocators). Moreover, they assume that no task within an atomic action is aborted; if the real-time language supports an abort facility then a task could be asynchronously removed from the action leaving the action in an inconsistent state.

In general, none of the mainstream languages or operating systems directly supports backward or forward error recovery facilities in the context of atomic actions. (However, C/Real-Time POSIX, Ada and Java do provide asynchronous notification mechanisms which can be used to help program recovery – see Sections 7.5, 7.6 and 7.7.) Language mechanisms have been proposed in research-oriented systems. In order to discuss these mechanisms, a simple language framework for atomic actions is introduced. The proposed recovery mechanisms are then discussed in the context of this framework.

To simplify the framework, only static tasks will be considered. Also it will be assumed that all the tasks taking part in an atomic action are known at compile-time. Each task involved in an action declares an action statement which specifies: the action name, the other tasks taking part in the action, and the code to be executed by the declaring task on entry to the action. For example a task P_1 which wishes to enter into an atomic action A with tasks P_2 and P_3 would declare the following action:

```

action A with ( $P_2$ ,  $P_3$ ) do
    -- acquire resources
    -- communicate with  $P_2$  and  $P_3$ 
    -- release resources
end A;

```


It is assumed that resource allocators are known and that communication inside the action is restricted to the three P tasks (together with external calls to the resource allocators). These restrictions are checked at compile-time. All other tasks declare similar actions, and nested actions are allowed as long as strict nesting is observed. Note that if the tasks are not known at compile-time, then any communication with a task will be allowed only if both tasks are active in the same atomic action.

The imposed synchronization on the action is as follows. Tasks entering the action are not blocked. A task is blocked inside the action only if it has to wait for a resource to be allocated, or if it attempts to communicate with another task inside the action and that task is either active in the action, but not in a position to accept the communication, or is not as yet active in the action.

Tasks may leave the action only when all tasks active in the action wish to leave. This was not the case in the examples given earlier. There it was assumed that all tasks must enter the action before any could leave. Here it is possible for a subset of the named tasks to enter the action and subsequently leave (without recourse to any interactions with the missing tasks). This facility is deemed to be essential in a real-time system where deadlines are important. It solves the **deserter** problem where all tasks are held in an action because one task has not arrived. This issue will be considered along with error recovery in the next two subsections.

7.3.1 Atomic actions and backward error recovery

Atomic actions are important because they constrain the flow of information around the system to well-defined boundaries and therefore can provide the basis for both damage confinement and error recovery. In this section, backward error recovery between concurrent tasks is described.

In Chapter 2, it was shown that when backward error recovery is applied to groups of communicating tasks, it is possible for all the tasks to be rolled back to the start of their execution. This was the so-called *domino effect*. The problem occurred because there was no consistent set of recovery points or a recovery line. An atomic action provides that recovery line automatically. If an error occurs inside an atomic action then the tasks involved can be rolled back to the start of the action and alternative algorithms executed; the atomic action ensures that tasks have not passed any erroneous values through communication with tasks outside the action. When atomic actions are used in this way they are called **conversations** (Randell, 1975).

With conversations each action statement contains a recovery block. For example:

```
action A with ( $P_2$ ,  $P_3$ ) do
  ensure <acceptance test>
  by
    -- primary module
  else by
    -- alternative module
  else by
    -- alternative module
  else error
end A;
```


Other tasks involved in the conversation declare their part in the action similarly. The basic semantics of a conversation can be summarized as follows.

- On entry to the conversation, the state of a task is saved. The set of entry points forms the recovery line.
- While inside the conversation, a task is allowed only to communicate with other tasks active in the conversation and general resource managers. As conversations are built from atomic actions, this property is inherited.
- In order to leave the conversation, all tasks active in the conversation must have passed their acceptance test. If this is the case, then the conversation is finished and all recovery points are discarded.
- *If any task fails its acceptance test, all tasks have their state restored to that saved at the start of the conversation and they execute their alternative modules.* It is, therefore, assumed that any error recovery to be performed inside a conversation *must* be performed by *all* tasks taking part in the conversation.
- Conversations can be nested, but only strict nesting is allowed.
- If all alternatives in the conversation fail then recovery must be performed at a higher level.

It should be noted that in conversations, as defined by Randell (1975), all tasks taking part in the conversation must have entered the conversation before any of the other tasks can leave. This differs from the semantics described here. If a task does not enter into a conversation, either because of tardiness or because it has failed, then as long as the other tasks active in the conversation do not wish to communicate with it, the conversation can complete successfully. If a task does attempt to communicate with a missing task then either it can block and wait for the task to arrive or it can continue. Adopting this approach has two benefits (Gregory and Knight, 1985).

- It allows conversations to be specified where participation is not compulsory.
- It allows tasks with deadlines to leave the conversation, continue and if necessary take some alternative action.

Although conversations allow groups of tasks to coordinate their recovery, they have been criticized. One important point is that when a conversation fails, all the tasks are restored and all enter their alternative modules. This forces the same tasks to communicate again to achieve the desired effect; a task cannot break out of the conversation. This may be not what is required. Gregory and Knight (1985) point out that in practice when one task fails to achieve its goal in a primary module through communication with one group of tasks, it may wish to communicate with a completely new group of tasks in its secondary module. Furthermore, the acceptance test for this secondary module may be quite different. There is no way to express these requirements using conversations.

7.3.2 Atomic actions and forward error recovery

It was pointed out in Chapter 2 that, although backward error recovery enables recovery from unanticipated errors, it is difficult to undo any operation that may have been

performed in the environment in which the embedded system operates. Consequently forward error recovery and exception handling must be considered. In this section, exception handling between the concurrent tasks involved in an atomic action is discussed.

With backward error recovery, when an error occurs all tasks involved in the atomic action participate in recovery. The same is true with exception handling and forward error recovery. If an exception occurs in one of the tasks active in an atomic action then that exception is raised in *all* tasks active in the action. The exception is said to be **asynchronous** as it originates from another task. The following is a possible Ada-like syntax for an atomic action supporting exception handling:

```

action A with (P2, P3) do
  -- the action
exception
  when exception_a =>
    -- sequence of statements
  when exception_b =>
    -- sequence of statements
  when others =>
    raise atomic_action_failure;
end A;

```

With the termination model of exception handling, if all tasks active in the action have a handler and all handle the exception without raising any further exception, then the atomic action completes normally. If a resumption model is used, when the exception has been handled, the tasks active in the atomic action resume their execution at the point where the exception was raised.

With either model, if there is no exception handler *in any one of the tasks active in the action* or one of the handlers fails then *the atomic action fails* with a standard exception `atomic_action_failure`. This exception is raised in all the involved tasks.

There are two issues which must be considered when exception handling is added to atomic actions: resolution of concurrently raised exceptions and exceptions in nested actions (Campbell and Randell, 1986). These are now briefly reviewed.

Resolution of concurrently raised exceptions

It is possible for more than one task active in an atomic action to raise different exceptions at the same time. As Campbell and Randell (1986) point out, this event is likely if the errors resulting from some fault cannot be uniquely identified by the error-detection facility provided by each component of the atomic action. If two exceptions are simultaneously raised in an atomic action, there may be two separate exception handlers in each task. It may be difficult to decide which one should be chosen. Furthermore, the two exceptions in conjunction constitute a third exception which is the exception that indicates that both the other two exceptional conditions have occurred.

In order to resolve concurrently raised exceptions, Campbell and Randell propose the use of an **exception tree**. If several exceptions are raised concurrently then the exception used to identify the handler is that at the root of the smallest subtree that contains all the exceptions (although it is not clear how to combine any parameters

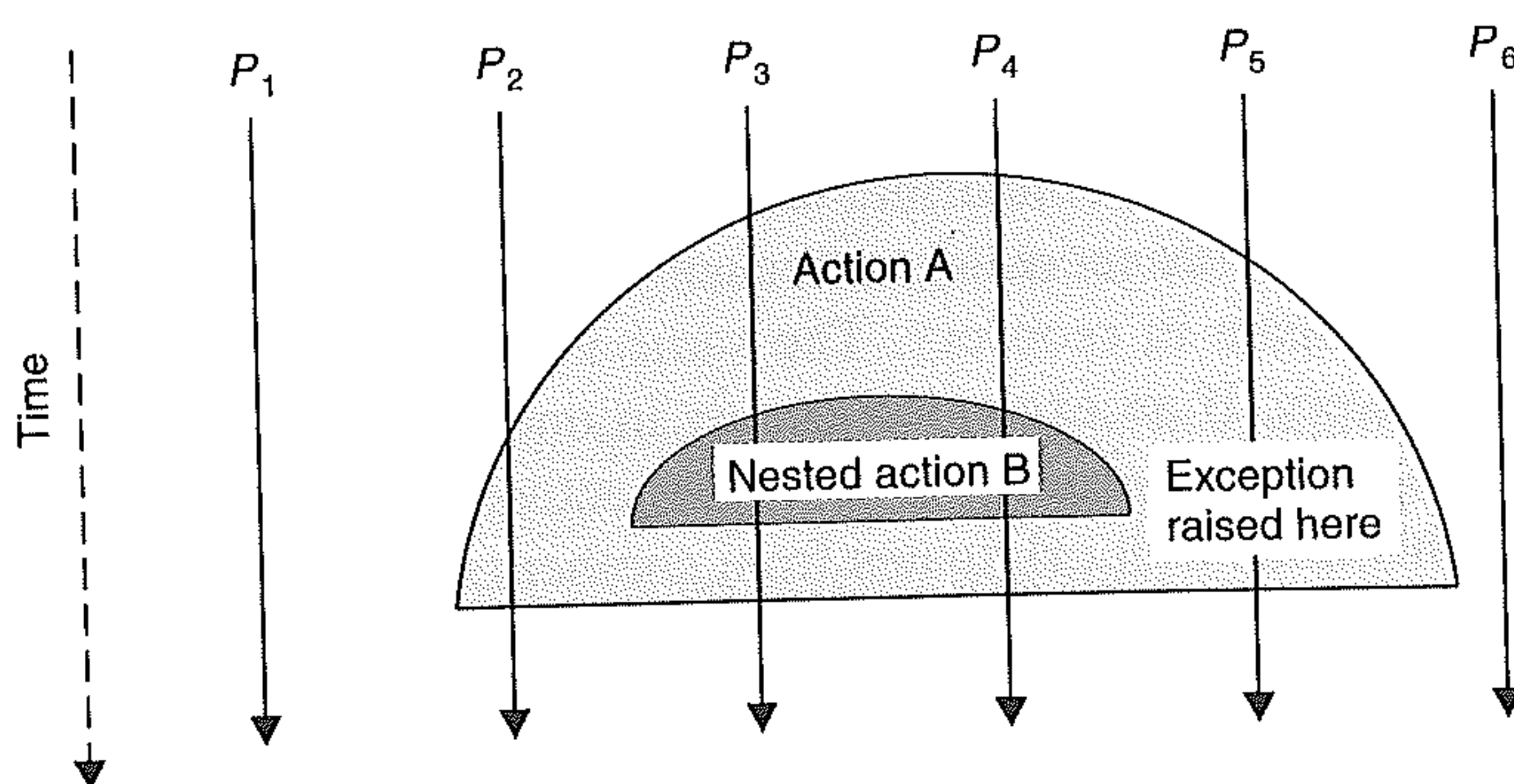


Figure 7.4 An exception in a nested atomic action.

associated with this exception). Each atomic action component can declare its own exception tree; the different tasks involved in an atomic action may well have different exception trees.

Exceptions and internal atomic actions

Where atomic actions are nested, it is possible for one task active in an action to raise an exception when other tasks in the same action are involved in a nested action. Figure 7.4 illustrates the problem.

When the exception is raised, all tasks involved must participate in the recovery action. Unfortunately, the internal action, by definition, is indivisible. To raise the exception in that action would potentially compromise that indivisibility. Furthermore, the internal action may have no knowledge of the possible exceptions that can be raised.

Campbell and Randell (1986) have discussed two possible solutions to this problem. The first solution is to hold back the raising of the exception until the internal action has finished. This they reject because:

- In a real-time system, the exception being raised may be associated with the missing of a deadline. To hold back the recovery procedure may seriously place in jeopardy the action's timely response.
- The error condition detected may indicate that the internal action may never terminate because some deadlock condition has arisen.

For these reasons, Campbell and Randell allow internal actions to have a predefined abortion exception. This exception is raised to indicate to the action that an exception has been raised in a surrounding action and that the preconditions under which the action was invoked are no longer valid. If such an exception is raised, the internal action should invoke fault-tolerant measures to abort itself. Once the action has been aborted, the containing action can handle the original exception.

If the internal action cannot abort itself, then it must signal an atomic action failure exception. This then may be combined with the outstanding exception so as to affect the choice of recovery performed by the surrounding action. If no abortion exception is

defined, the surrounding action must wait for the internal action to complete. Alternatively, a default handler could be provided which would raise the atomic action failure exception.

7.4 Asynchronous notification

Although forward and backward error recovery have been discussed separately, in reality they may need to be combined in many real-time systems. Backward error recovery is needed to recover from unanticipated errors, and forward error recovery is needed to undo or ameliorate any interaction with the environment. Indeed, forward error handling can be used to implement a backward error recovery scheme – see Section 7.6.3.

As discussed in Section 7.2, none of the major real-time languages supports atomic actions, and it is necessary to use more primitive language facilities to achieve the same effect. The same is true for recoverable actions. One of the main requirements for a recoverable action is to be able to gain the attention of a task involved in an action and notify it that an error has occurred in another task. Most languages and operating systems support some form of asynchronous notification mechanism. As with exceptions, there are two basic models: resumption and termination.

The resumption model of asynchronous notification handling (often called **event handling**) behaves like a software interrupt. A task indicates which events it is willing to handle; when the event is signalled, the task is interrupted (unless it has temporarily inhibited the event from being delivered) and an event handler is executed. The handler responds to the asynchronous event and then the task continues with its execution from the point at which it was interrupted. This, of course, sounds very similar to the resumption model of exception handling given in Section 3.2.4. The main difference is that the event is usually *not* signalled by the affected task (or because of an operation the affected task is performing), but is signalled *asynchronously*. However, many operating systems do not provide a special exception-handling facility for synchronous exception handling, but use the asynchronous events mechanisms instead. The C/Real-Time POSIX signal facility is an example of an asynchronous event model with resumption.

Note that, with the resumption model, the flow of control of a task is only temporarily changed; after the event has been handled the task is resumed. In a multithreaded process, it is possible to associate a distinct thread with the event and to schedule the thread when the event is signalled. Real-Time Java provides support for this model.

With the termination model of asynchronous notification, each task specifies a domain of execution during which it is prepared to receive an asynchronous notification that will cause the domain to be terminated. This form is often called **asynchronous transfer of control** or ATC. If an ATC request occurs outside this domain, it may be ignored or queued. After the ATC has been handled, control is returned to the interrupted task at a location different from that where the ATC was delivered. This, of course, is very similar to the termination model of exception handling. The Ada and Real-Time Java languages support asynchronous transfer of control mechanisms.

An extreme form of asynchronous notification with termination semantics is to abort the task and allow another task to perform some recovery. All operating systems and most concurrent programming languages provide such a facility. However, aborting

a process can be expensive and is often an extreme response to many error conditions. Aborting a thread is less expensive but still potentially dangerous as it can leave resources in an undefined state. Consequently, some form of safe asynchronous notification mechanism is also required.

The inclusion of an asynchronous notification mechanism into a language (or operating system) is controversial, as it complicates the language's semantics and increases the complexity of the run-time support system. This section thus first considers the application requirements which justify the inclusion of such a facility. The C/Real-Time POSIX, Ada and Real-Time Java models of asynchronous notification are then discussed.

7.4.1 The user need for asynchronous notification

The fundamental requirement for an asynchronous notification facility is to enable a task to respond *quickly* to a condition which has been detected by another task. The emphasis here is on a quick response; clearly a task can always respond to an event by simply polling or waiting for that event. The notification of the event could be mapped onto the task's communication and synchronization mechanism. The handling task, when it is ready to receive the event, simply issues the appropriate request.

Unfortunately, there are occasions when polling for events or waiting for the event to occur is inadequate. These include the following.

- **Error recovery** – this chapter has already emphasized that when groups of tasks undertake atomic actions, an error detected in one task requires all other tasks to participate in the recovery. For example, a hardware fault may mean that the task will never finish its planned execution because the preconditions under which it started no longer hold; the task may never reach its polling point. Also, a timing fault might have occurred, which means that the task will no longer meet the deadline for the delivery of its service. In both these situations, the task must be informed that an error has been detected and that it must undertake some error recovery as quickly as possible.
- **Mode changes** – a real-time system often has several modes of operation. For example, a fly-by-wire civil aircraft may have a take-off mode, a cruising mode and a landing mode. On many occasions, changes between modes can be carefully managed and will occur at well-defined points in the system's execution, as in a normal flight plan for a civil aircraft. Unfortunately, in some application areas, mode changes are expected but cannot be planned. For example, a fault may lead to an aircraft abandoning its take-off and entering an emergency mode of operation; or an accident in a manufacturing task may require an immediate mode change to ensure an orderly shutdown of the plant. In these situations, tasks must be quickly and safely informed that the mode in which they are operating has changed, and that they now need to undertake a different set of actions.
- **Scheduling using partial/imprecise computations** – there are many algorithms where the accuracy of the results depends on how much time can be allocated to their calculation. For example, numerical computations, statistical estimations and heuristic searches may all produce an initial estimation of the required result, and then refine that result to a greater accuracy. At run-time, a certain amount of time

can be allocated to an algorithm, and then, when that time has been used, the task must be interrupted to stop further refinement of the result.

- **User interrupts** – in a general interactive computing environment, users often wish to stop the current processing because they have detected an error condition and wish to start again.

7.5 Asynchronous notification in C/Real-Time POSIX

The C/Real-Time POSIX model supports two forms of asynchronous notification: a resumption facility based on signals, and a thread cancellation mechanism.

7.5.1 C/Real-Time POSIX signals

C/Real-Time POSIX signals support the resumption model. They are also used for a class of environment-detected synchronous errors (such as divide by zero, illegal pointer and so on).

There are a number of predefined signals, each of which is allocated an integer value. There are also an implementation-defined number of signals which are available for application use. Each signal has a default handler, which usually terminates the receiving process. Example signals are: `SIGABRT` for abnormal termination, `SIGALARM` for alarm clock expiry, `SIGILL` for illegal instruction exception, `SIGRTMIN` for the identifier of the first real-time application-definable exception, and `SIGRTMAX` for the identifier of the last real-time application-definable exception. Only those signals whose numbers lie between `SIGRTMIN` and `SIGRTMAX` are considered to be real-time by POSIX. A real-time signal is one which can have extra information passed to the handler by the process which generated it; in addition, they are queued.

The program can specify that it wants to receive signals as a result of certain events occurring; for example, when a message is received on a message queue (see the `mq_notify` function in Program 6.1) or a timer expires (see the `timer_create` function in Program 10.7). An application can configure its requirements for generating and receiving signals via the use of several data structures. These are shown in Program 7.1.

The `sigevent` structure is used to pass information to the signal-handling subsystem concerning the generation of a signal. For example, `timer_create` functions take an object of this type to indicate what should happen when the timer expires. The `sigev_signo` indicates the identity of the signal to be generated, and `sigev_value` indicates what data should be passed to the handler. The `sigev_notify` indicates how the program should be notified when the signal is generated. There are three possibilities.

- `SIGEV_NONE` – no notification occurs when the signal is generated.
- `SIGEV_SIGNAL` – a signal is queued with the application-defined data.
- `SIGEV_THREAD` – a new pthread should be created with a start routine (see `pthread_create` function in Program 4.5) defined by `sigev_notify_function` and thread attributes defined by `sigev_notify_attributes`.

Program 7.1 The C/Real-Time POSIX interface to data structures supporting signal generation.

```

/* used with message queue notification, timers etc */

struct sigevent {
    int sigev_notify;
    /* SIGEV_SIGNAL, */
    /* SIGEV_THREAD or SIGEV_NONE */

    int sigev_signo; /* signal to be generated */

    union sigval sigev_value; /* value to be queued */

    void (*)sigev_notify_function(union sigval s);
    /* function to be treated as thread */
    pthread_attr_t *sigev_notify_attributes;
    /* thread attributes */
};

union sigval {
    int sival_int;
    void *sival_ptr;
};

```

If a signal is to be delivered to the process itself rather than via creating a thread, then further types and functions are available in C/Real-Time POSIX to specify the program's response. These are shown in Program 7.2. Most of the predefined C/Real-Time POSIX signals are delivered in this manner.

There are three ways in which a process can deal with a signal.

- It can **block** the signal and either handle it later or accept it.
- It can **handle** the signal by setting a function to be called whenever it occurs.
- It can **ignore** the signal altogether (in which case the signal is simply lost).

A signal that is not blocked and not ignored is **delivered** as soon as it is **generated**. A signal that is blocked is **pending** delivery, or may be **accepted** by calling one of the `sigwait()` functions.

Blocking a signal

C/Real-Time POSIX maintains the set of signals that have been currently masked by a process. The function `sigprocmask` is used to manipulate this set. The `how` parameter is set to: `SIG_BLOCK` to add signals to the set, `SIG_UNBLOCK` to subtract signals from the set, or `SIG_SETMASK` to replace the set.¹ The other two parameters contain pointers

¹`SIG_BLOCK`, `SIG_UNBLOCK` and `SIG_SETMASK` are compile-time constants.

Program 7.2 An abridged C/Real-Time POSIX interface to signals.

```
typedef ... sigset_t;

/* the following manipulates the signal mask */
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
/* how = SIG_BLOCK -> the set is added to the current set */
/* how = SIG_UNBLOCK -> the set is subtracted from the */
/*      current set */
/* how = SIG_SETMASK -> the given set becomes the mask */

/* the following routines allow a signal */
/* set to be created and manipulated */
int sigemptyset(sigset_t *s); /* initialize a set to empty */
int sigfillset(sigset_t *s); /* initialize a set to full */
int sigaddset(sigset_t *s, int signum); /* add a signal */
int sigdelset(sigset_t *s, int signum); /* remove a signal */
int sigismember(const sigset_t *s, int signum);
/* returns 1 if a member */

/* the following support signal handling */

typedef struct { /* signal parameters */
    int si_signo;
    int si_code;
    union sigval si_value;
} siginfo_t;

struct sigaction {
    void (*sa_handler) (int signum); /* non real-time handler */
    void (*sa_sigaction) (int signum, siginfo_t *data,
                          void *extra); /* real-time handler */
    sigset_t sa_mask; /* signals to mask during handler */
    int sa_flags; /* indicates if signal is to be queued */
};

int sigaction(int sig, const struct sigaction *reaction,
              struct sigaction *old_reaction);
/* sets up a signal handler, reaction, for sig */

/* the following functions allow a */
/* process to wait for a signal */

int sigsuspend(const sigset_t *sigmask);
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
int sigtimedwait(const sigset_t *set, siginfo_t *info,
                 const struct timespec *timeout);

/* the following functions allow a */
/* signal to be sent */
int kill (pid_t pid, int sig);
/* send the signal sig to the process pid */
int sigqueue(pid_t pid, int sig, const union sigval value);
/* send signal and data */

/* All the above functions return -1 when errors have occurred. */
/* A shared variable errno contains the reason for the error */
```

to the set of signals to be added/subtracted/replaced (`set`) and the returned value of the old set (`oset`). Various functions (`sigemptyset`, `sigfillset`, `sigaddset`, `sigdelset` and `sigismember`) allow a set of signals to be manipulated.

When a signal is blocked, it remains pending until it is unblocked or accepted. When it is unblocked, it is then delivered. Note, some signals *cannot* be blocked.

Handling a signal

A signal handler can be set up using the function `sigaction`. The `sig` parameter indicates which signal is to be handled, `reaction` is a pointer to a structure containing information about the handler, and `old_reaction` points to information about the previous handler. Essentially, the information about a handler contains a pointer to the handler function (`sa_handler` if the signal is a non-real-time signal, or `sa_sigaction` if the signal is a real-time one), the set of signals to be masked during the handler's execution (`sa_mask`), and whether the signal is to be queued (indicated by setting `sa_flags` to the symbolic constant `SA_SIGINFO` – only signals whose value lies between `SIGRTMIN` and `SIGRTMAX` can be queued). The `sa_handler` member indicates the action associated with the signal and can be:

- `SIG_DFL` – default action (usually to terminate the process);
- `SIG_IGN` – ignore the signal;
- pointer to a function – to be called when the signal is delivered.

For non-real-time signals, only an integer parameter can be passed to the handler when the signal is generated. The value of this parameter normally indicates the signal itself (the same handler can be used for more than one signal). However, for the real-time signals, more data can be passed via a pointer to the `siginfo_t` structure. This structure contains the signal number (again), a code which indicates the cause of the signal (for example, a timer signal) and an integer or pointer value.

If more than one *real-time* signal is queued, the one with the lowest value is delivered first (that is, `SIGRTMIN` is delivered before `SIGRTMIN + 1` and so on).

A process can also wait for a signal to arrive using the functions `sigsuspend`, `sigwaitinfo` or `sigtimedwait`. The function `sigsuspend` replaces the mask with that given by the parameter to the call and suspends the process until:

- (1) a non-blocked signal is delivered, and
- (2) the associated handler is executed.

If the handler terminates the process, the `sigsuspend` function never returns; otherwise it returns with the signal mask reset to the state that existed *prior* to the call of `sigsuspend`.

The `sigwaitinfo` function also suspends the calling process until the signal arrives. However, this time the signal must be blocked, and thus the handler is not called. Instead, the function returns the selected signal number and stores the information about the delivered signal in the `info` argument. The function `sigtimedwait` has the same

semantics as `sigwaitinfo`, but allows a timeout to be specified for the suspension. If no signals are delivered by the timeout, `sigwaitinfo` returns with `-1` and `errno` set to `EAGAIN`.

Care must clearly be taken when using signals for condition synchronization. There is a potential race condition between checking to see whether a signal has already arrived and issuing a request which causes suspension. The appropriate protocol is to block the signal first, then test to see if it has occurred and, if not, suspend and unblock the signal using one of the above functions.

Ignoring a signal

A signal can be ignored by simply setting the value of `sa_handler` to `SIG_IGN` in a call to the function `sigaction`.

Generating a signal

There are two ways in which a process can generate a signal to be sent to another process. The first is via the `kill` function and the second is via the `sigqueue` function. The latter can only send real-time signals.

Note, however, that a process can also request that a signal be sent to itself: when a timer expires (for example, `SIGALRM` – see Section 13.2.3), when asynchronous I/O completes, by the arrival of a message on an empty message queue (see Section 6.7), or by using the C `raise` statement.

A simple example of C/Real-Time POSIX signals and processes

As an illustration of C/Real-Time POSIX signals, consider the program fragment below. A process performs some computation periodically. The actual computation to be performed depends on a system-wide mode of operation. A mode change is propagated to all processes via an application-defined real-time signal `MODE_CHANGE`. The signal handler `change_mode` simply changes a global variable `mode`. The processes access `mode` at the beginning of each iteration. To ensure that the mode does not change while it is accessing it, the `MODE_CHANGE` signal is blocked.

```
#include <signal.h>

#define MODE_A 1
#define MODE_B 2
#define MODE_CHANGE SIGRTMIN + 1

int mode = MODE_A;

void change_mode(int signum, siginfo_t *data, void *extra) {
    /* signal handler */
    mode = data -> si_value.sival_int;
}
```

```

int main() {

    sigset_t mask, omask;
    struct sigaction s, os;
    int local_mode;

    SIGEMPTYSET(&mask);
    SIGADDSET(&mask, MODE_CHANGE);

    s.sa_flags = SA_SIGINFO;
    s.sa_mask = mask;
    s.sa_sigaction = &change_mode;
    s.sa_handler = &change_mode;

    SIGACTION(MODE_CHANGE, &s, &os); /* assign handler */

    while(1) {

        SIGPROCMASK(SIG_BLOCK, &mask, &omask);
        local_mode = mode;
        SIGPROCMASK(SIG_UNBLOCK, &mask, &omask);

        /* periodic operation using mode*/
        switch(local_mode) {
            case MODE_A:
                ...
                break;
            case MODE_B:
                ...
                break;
            default:
                ...
        }
    }
}

```

Signals and threads

The original C/POSIX signal model came from Unix and was extended to make it more appropriate for real-time when the Real-Time Extensions to POSIX were specified. With the POSIX Thread Extensions, the model has become more complex and is a compromise between a per process model and a per thread model. The following points should be noted.

- Signals which are generated as a result of a synchronous error condition, such as memory violation, are delivered only to the thread that caused the signal.
- Other signals may be sent to the process as a whole; however, each one is only delivered to a single thread in the process.
- The `sigaction` function sets the handler for *all* threads in the process.

- The functions `kill` and `sigqueue` still apply to processes. A new function `pthread_kill`

```
int pthread_kill(pthread_t thread, int sig);
```

allows a process to send a signal to an individual thread.

- If more than one thread is eligible to have a signal delivered to it, it is not defined which thread is chosen.
- If the action specified by a handler for the signal is for termination, *the whole process is terminated*, not just the thread.
- Signals can be blocked on a per thread basis using a function `pthread_sigmask` which has the same set of parameters as `sigprocmask`. The use of the function `sigprocmask` is not specified for a multithreaded process.
- The functions `sigsuspend`, `sigwaitinfo` or `sigtimedwait` operate on the calling thread not the calling process.
- A new function `sigwait`

```
int sigwait(const sigset_t *set, int *sig);
```

allows a thread to wait for one of several blocked signals to occur. It behaves the same as `sigwaitinfo()` except that the information associated with the signal is not returned. The signals are specified in the location referenced `set`. The function returns zero when a successful wait has been performed and the location referenced by `sig` contains the received signal.

If one of the signals is already pending when the function is called, the function returns immediately. If more than one is pending, it is not defined which one is chosen unless only real-time signals are pending. In this case, the one with the lowest value is chosen.

- If a signal action is set by a thread to 'ignore', it is unspecified whether the signal is discarded immediately it is generated or remains pending.

Although C/Real-Time POSIX allows a thread or a process to handle an asynchronous event, care must be taken because some of the POSIX system calls are termed **async-signal unsafe** and **async-cancel unsafe**. It is undefined what happens if a signal interrupts an async-unsafe function that was called from a signal-catching function. For example, it is not safe to use the function `pthread_cond_signal` in a signal handler because of the race condition it introduces with the function `pthread_cond_wait`.

7.5.2 Asynchronous transfer of control and thread cancellation

C/Real-Time POSIX supports the C `setjmp` and `longjmp` mechanisms. As discussed in Section 3.3.3 these can be used to implement a termination model of exception handling. There are also versions of these that can be used in conjunction with signals (`sigsetjmp` and `siglongjmp`). These as well as saving the thread's state, save and restore the signal mask respectively. Hence it is conceivable that an ATC could be

Program 7.3 The C/Real-Time POSIX interface to thread cancellation.

```

#define PTHREAD_CANCEL_ASYNCHRONOUS ...
#define PTHREAD_CANCEL_ENABLE ...
#define PTHREAD_CANCEL_DEFERRED ...
#define PTHREAD_CANCEL_DISABLE ...
#define PTHREAD_CANCELED ...

int pthread_cancel(pthread_t thread);
void pthread_cleanup_push(void (* routine)(void *), void * arg);
void pthread_cleanup_pop(int execute);

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);

```

implemented using a combination of these mechanisms. Unfortunately, these non-local jump functions and all the thread system calls are async-signal unsafe. C/Real-Time POSIX does not define what happens when an async-signal unsafe function is interrupted by a signal handler and the handler calls an async-signal unsafe function.

Threads can also be asynchronously terminated using the C/Real-Time POSIX thread cancellation facilities. However, a thread can control if and when it is cancelled via its cancellation state and cancellation type. The state can be **enabled** or **disabled**. If disabled, any attempt to cancel the thread is held pending. If enabled, cancellation will occur at point determined by the cancellation type. If the type is **asynchronous**, the cancellation may occur at any time. If **deferred**, the cancellation will occur at the next cancellation point that the thread reaches in its execution. The defaults for a newly created thread are enabled and deferred.

The full set of cancellation points is defined in the C/Real-Time POSIX standard. However, it is worth noting that most of the C/Real-Time POSIX pthread functions are *not* cancellation points.

In support of safe cancellation, C/Real-Time POSIX also allows a thread to define one or more routines that will be executed when cancellation is acted upon. These routines are stored on a stack, and the application can push and pop them during its execution. On cancellation (or when `pthread_exit` is called), they are popped off (in LIFO order) and executed.

Program 7.3 gives the full C/Real-Time POSIX API that supports cancellation and cleanup routines. Note that a thread cancels another thread by calling the function `pthread_cancel`, and a thread can introduce a cancellation point into its execution by calling `pthread_testcancel`.

7.5.3 C/Real-Time POSIX and atomic actions

Given the close interaction between activities in an atomic action, it is more appropriate to consider the action to take place between C/Real-Time POSIX threads rather than

C/Real-Time POSIX processes. Atomic actions are easiest to implement if a termination model of asynchronous transfer of control is used.

Given the problems of using a combination of `setjmp/longjmp` and signals, it is best to use thread creation and cancellation to program the required recovery. As C/Real-Time POSIX threads are designed to be efficient, this approach does not have the same performance penalty that would be associated with a more heavyweight process structure.

7.6 Asynchronous notification in Ada

The asynchronous notification facilities in Ada allow an application to respond to:

- events being signalled asynchronous from the external environment – this is in support of interrupt handling and will be considered in detail in Section 14.3;
- events being triggered by the passage of time – the handling for these events is executed at the priority of the clock device and they are considered in detail in Section 10.4.1;
- asynchronous transfer of control (ATC) requests on a task – supporting a termination model;
- task abortion.

There is no generalized mechanisms for a resumption model of asynchronous notification, hence this section will focus on ATC and task abortion.

7.6.1 Asynchronous transfer of control

Ada provides a structured form of asynchronous notification handling called **asynchronous transfer of control** (ATC). To emphasize that ATC is a form of communication and synchronization, the mechanism is built on top of the inter-task communication facility.

The Ada **select** statement was introduced in Chapter 6. It has the following forms:

- a selective accept (to support the server side of the rendezvous) – this was discussed in Section 6.5;
- a timed and a conditional entry call (to either a task or a protected entry) – this is discussed in Section 9.4.2;
- an asynchronous select – discussed here.

The asynchronous select statement provides an asynchronous notification mechanism with termination semantics.

The execution of the asynchronous select begins with the issuing of the triggering entry call or the issuing of the triggering delay. If the triggering statement is an entry call, the parameters are evaluated as normal and the call issued. If the call is queued, then a sequence of statements in an abortable part is executed.

If the triggering statement completes before the execution of the abortable part completes, the abortable part is aborted. When these activities have finished, the optional sequence of statements following the triggering statement is executed.

If the abortable part completes before the completion of the entry call, an attempt is made to cancel the entry call and, if successful, the execution of the asynchronous select statement is finished. The following illustrates the syntax:

```
select
  Trigger.Event;
  -- optional sequence of statements to be
  -- executed after the event has been received
then abort
  -- abortable sequence of statements
end select;
```

Note that the triggering statement can be a delay statement and, therefore, a timeout can be associated with the abortable part (see Section 9.4.3).

If the cancellation of the triggering event fails because the protected action or rendezvous has started, then the asynchronous select statement waits for the triggering statement to complete before executing the optional sequence of statements following the triggering statement.

Clearly, it is possible for the triggering event to occur even before the abortable part has started its execution. In this case the abortable part is not executed and therefore not aborted.

Consider the following example:

```
task Server is
  entry Atc_Event;
end Server;

task To_Be_Interrupted;

task body Server is
begin
  ...
  accept Atc_Event do
    Seq2;
  end Atc_Event;
  ...
end Server;

task body To_Be_Interrupted is
begin
  ...
  select -- ATC statement
    Server.Atc_Event;
    Seq3;
  then abort
    Seq1;
  end select;
  Seq4;
  ...
end To_Be_Interrupted;
```


When the above ATC statement is executed, the statements which are executed will depend on the order of events that occur:

```

if the rendezvous is available immediately then
    Server.Atc_Event is issued
    Seq2 is executed
    Seq3 is executed
    Seq4 is executed (Seq1 is never started)
elsif no rendezvous starts before Seq1 finishes then
    Server.Atc_Event is issued
    Seq1 is executed
    Server.Atc_Event is cancelled
    Seq4 is executed
elsif the rendezvous finishes before Seq1 finishes then
    Server.Atc_Event is issued
    partial execution of Seq1 occurs concurrently with Seq2
    Seq1 is aborted and finalized
    Seq3 is executed
    Seq4 is executed
else (the rendezvous finishes after Seq1 finishes)
    Server.Atc_Event is issued
    Seq1 is executed concurrently with partial execution of Seq2
    Server.Atc_Event cancellation is attempted but is unsuccessful
    execution of Seq2 completes
    Seq3 is executed
    Seq4 is executed
end if

```

Note that there is a race condition between Seq1 finishing and the rendezvous finishing. The situation could occur where Seq1 does finish but is nevertheless aborted.

Ada allows some operations to be **abort deferred**. If Seq1 contains an abort-deferred operation, then its cancellation will not occur until the operation is completed. An example of such an operation is a call on a protected object.

The above discussion has concentrated on the concurrent behaviour of Seq1 and the triggering rendezvous. Indeed, on a multiprocessor implementation it could be the case that Seq1 and Seq2 are executing in parallel. However, on a single-processor system, the triggering event will only ever occur if the action that causes it has a higher priority than Seq1. The normal behaviour will thus be the preemption of Seq1 by Seq2. When Seq2 (the triggering rendezvous) completes, Seq1 will be aborted before it can execute again. And hence the ATC is 'immediate' (unless an abort-deferred operation is in progress).

Exceptions and ATC

With the asynchronous select statement, potentially two activities occur concurrently: the abortable part may execute concurrently with the triggering action (when the action is an entry call). In either one of these activities, exceptions may be raised and unhandled. Therefore, at first sight it may appear that potentially two exceptions can be propagated simultaneously from the select statement. However, this is not the case: one of the exceptions is deemed to be lost (that raised in the abortable part when it is aborted), and hence only one exception is propagated.

7.6.2 Task abortion

Tasks in Ada can be aborted using an abort statement; any task may abort any other named task by executing this statement

Once aborted tasks are said to become *abnormal*, and are prevented from interacting with any other task. Ideally, an abnormal task will stop executing immediately. However, some implementations may not be able to facilitate immediate shut down, and hence all Ada requires is that the task terminate before it next interacts with other tasks. Note that the Real-Time Systems Annex does require ‘immediate’ to be just that on a single processor system.

After a task has been marked as abnormal, execution of its body is aborted. This means that the execution of every construct in the task body is aborted, unless it is involved in the execution of an *abort-deferred operation*. The execution of an abort-deferred operation is allowed to complete before it is aborted. The same rules for aborting a task body also apply to aborting a sequence of statements in the asynchronous select statement discussed earlier.

If a construct, which has been aborted, is blocked outside an abort-deferred operation (other than at an entry call), the construct becomes abnormal and is immediately completed. Other constructs must complete no later than the next *abort completion point* (if any) that occurs outside an abort-deferred operation.

An abort completion point occurs at:

- the end of activation of a task;
- the point where the execution initiates the activation of another task;
- the start or end of an entry call, accept statement, delay statement or abort statement;
- the start of the execution of a select statement, or of the sequence of statements of an exception handler.

The following operations are defined to be abort-deferred:

- a protected action;
- waiting for an entry call to complete;
- waiting for termination of dependent tasks;
- the execution of an ‘initialize’ procedure, a ‘finalize’ procedure, or an assignment operation of an object with a controlled part.

In Ada, the equivalent of C/Real-Time POSIX’s cleanup handlers can be programmed using `Controlled` types. Objects of these types must have their finalization routines executed when the task is aborted.

7.6.3 Ada and atomic actions

It was shown in Section 3.4 that backward error recovery in a sequential system could be implemented by exception handling. In this section, the Ada ATC facility and exception handling is used to implement backward and forward error recovery. It is assumed that the underlying Ada implementation and run-time are fault free, and therefore the strong typing provided by Ada will ensure that the Ada program itself remains viable.

Backward error recovery

The following package is a generic version of the one was given in Section 3.4 for saving and restoring a task's state.

```
generic
  type Data is private;
package Recovery_Cache is
  procedure Save(D : in Data);
  procedure Restore(D : out Data);
end Recovery_Cache;
```

Consider three Ada tasks which wish to enter into a recoverable atomic action. Each will call their appropriate procedure in the package given below:

```
package Conversation is

  procedure T1(Params : Param); -- called by task 1
  procedure T2(Params : Param); -- called by task 2
  procedure T3(Params : Param); -- called by task 3

  Atomic_Action_Failure : exception;

end Conversation;
```

The body of the package encapsulates the action and ensures that only communication between the three tasks is allowed during the conversation.² The `Controller` protected object is responsible for propagating any error condition noticed in one task to all tasks, saving and restoring any persistent data in the recovery cache, and ensuring that all tasks leave the action at the same time. It contains three protected entries and a protected procedure.

- The `Wait_Abort` entry represents the asynchronous event on which the tasks will wait while performing their part of the action.
- Each task calls `Done` if it has finished its component of the action without error. Only when all three tasks have called `Done` will they be allowed to leave.
- Similarly, each task calls `Cleanup` if it has had to perform any recovery.
- If any task recognizes an error condition (either because of a raised exception or the failure of the acceptance test), it will call `Signal_Abort`. This will set the flag `Killed` to `True`, indicating that the tasks must be recovered.

Note that, as backward error recovery will be performed, the tasks are not concerned with the actual cause of the error. When `Killed` becomes `True`, all tasks in the action receive the asynchronous event. Once the event has been handled, all tasks must wait on the `Cleanup` entry so that they can all terminate the conversation module together.

²In practice, this might be difficult to ensure because of Ada's scope rules. One way of increasing the security would be to require that the `Conversation` package is at the library level and its body only references pure (state-free) packages. The solution presented here assumes that the tasks are well behaved. It also assumes, for simplicity, that the correct tasks call `T1`, `T2` and `T3` at the correct times.

```

with Recovery_Cache;
package body Conversation is

    Primary_Failure, Secondary_Failure,
        Tertiary_Failure: exception;
    type Module is (Primary, Secondary, Tertiary);

    protected Controller is
        entry Wait_Abort;
        entry Done;
        entry Cleanup;
        procedure Signal_Abort;
    private
        Killed : Boolean := False;
        Releasing_Done : Boolean := False;
        Releasing_Cleanup : Boolean := False;
        Informed : Integer := 0;
    end Controller;

    -- any local protected objects for communication between actions

    protected body Controller is
        entry Wait_Abort when Killed is
        begin
            Informed := Informed + 1;
            if Informed = 3 then
                Killed := False;
                Informed := 0;
            end if;
        end Wait_Abort;

        procedure Signal_Abort is
        begin
            Killed := True;
        end Signal_Abort;

        entry Done when Done'Count = 3 or Releasing_Done is
        begin
            if Done'Count > 0 then
                Releasing_Done := True;
            else
                Releasing_Done := False;
            end if;
        end Done;

        entry Cleanup when Cleanup'Count = 3 or Releasing_Cleanup is
        begin
            if Cleanup'Count > 0 then
                Releasing_Cleanup := True;
            else
                Releasing_Cleanup := False;
            end if;
        end Cleanup;
    end Controller;

```



```

procedure T1(Params : Param) is separate;
procedure T2(Params : Param) is separate;
procedure T3(Params : Param) is separate;

```

```

end Conversation;

```

The code for each task is contained within a single procedure: e.g. T1. Within such a procedure, three attempts are made to perform the action. If all attempts fail, the exception `Atomic_Action_Failure` is raised. Each attempt is surrounded by a call that saves the state and restores the state (if the attempt fails). Each attempt is encapsulated in a separate local procedure (`T1_Primary`, etc.), which contains a single 'select and then abort' statement to perform the required protocol with the controller. The recovery cache is used by each task to save its local data.

```

separate(Conversation)
procedure T1(Params : Param) is
  procedure T1_Primary is
    begin
      select
        Controller.Wait_Abort; -- triggering event
        Controller.Cleanup; -- wait for all to finish
        raise Primary_Failure;
      then abort
        begin
          -- code to implement atomic action,
          -- the acceptance test might raise an exception
          if Accept_Test = Failed then
            Controller.Signal_Abort;
          else
            Controller.Done; -- signal completion
          end if;
        exception
          when others =>
            Controller.Signal_Abort;
        end;
      end select;
    end T1_Primary;

  procedure T1_Secondary is ... ;
  procedure T1_Tertiary is ... ;

  package My_Cache is new Recovery_Cache(...); -- for local data

begin
  My_Cache.Save(...);
  for Try in Module loop
    begin
      case Try is
        when Primary => T1_Primary; return;
        when Secondary => T1_Secondary; return;
        when Tertiary => T1_Tertiary;
      end case;
    end
  end

```

```
exception
  when Primary_Failure =>
    My_Cache.Restore(..);
  when Secondary_Failure =>
    My_Cache.Restore(..);
  when Tertiary_Failure =>
    My_Cache.Restore(..);
    raise Atomic_Action_Failure;
  when others =>
    My_Cache.Restore(..);
    raise Atomic_Action_Failure;
end;
end loop;
end T1;

-- similarly for T2 and T3
```

Figure 7.5 illustrates a simple state transition diagram for a participating task in a conversation.

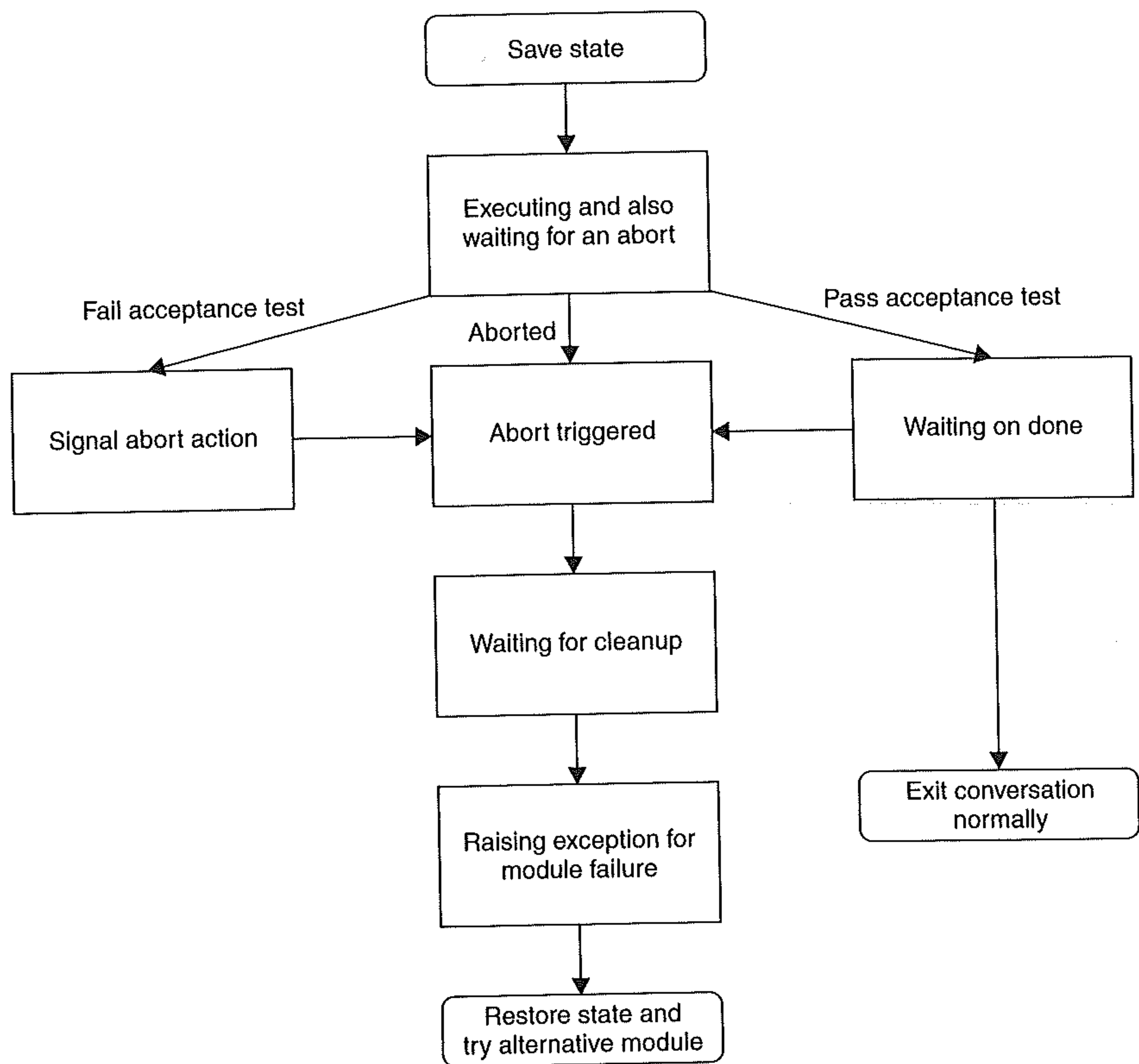


Figure 7.5 Simple state transition diagram for a conversation.

Forward error recovery

Ada's ATC facility can be used with exceptions to implement atomic actions with forward error recovery between concurrently executing tasks. Consider again the following package for implementing an atomic action between three tasks:

```
package Action is
  procedure T1(Params : Param); -- called by task 1
  procedure T2(Params : Param); -- called by task 2
  procedure T3(Params : Param); -- called by task 3

  Atomic_Action_Failure : exception;
end Action;
```

As with backward error recovery, the body of the package encapsulates the action and ensures that only communications between the three tasks are allowed. The Controller protected object is responsible for propagating any exception raised in one task to all tasks, and for ensuring that all tasks leave the action at the same time.

```
with Ada.Exceptions;
use Ada.Exceptions;
package body Action is
  type Vote_T is (Commit, Aborted);
  protected Controller is
    entry Wait_Abort(E: out Exception_Id);
    entry Done;
    procedure Cleanup (Vote: Vote_T);
    entry Wait_Cleanup(Result : out Vote_T);
    procedure Signal_Abort(E: Exception_Id);
  private
    Killed : Boolean := False;
    Releasing_Cleanup : Boolean := False;
    Releasing_Done : Boolean := False;
    Reason : Exception_Id;
    Final_Result : Vote_T := Commit;
    Informed : Integer := 0;
  end Controller;

  -- any local protected objects for communication between actions

  protected body Controller is
    entry Wait_Abort(E: out Exception_Id) when Killed is
    begin
      E := Reason;
      Informed := Informed + 1;
      if Informed = 3 then
        Killed := False;
        Informed := 0;
      end if;
    end Wait_Abort;

    entry Done when Done'Count = 3 or Releasing_Done is
    begin
```

```

if Done'Count > 0 then
    Releasing_Done := True;
else
    Releasing_Done := False;
end if;
end Done;

procedure Cleanup (Vote: Vote_T) is
begin
    if Vote = Aborted then
        Final_Result := Aborted;
    end if;
end Cleanup;

procedure Signal_Abort(E: Exception_Id) is
begin
    Killed := True;
    Reason := E;
end Signal_Abort;

entry Wait_Cleanup (Result: out Vote_T)
    when Wait_Cleanup'Count = 3 or Releasing_Cleanup is
begin
    Result := Final_Result;
    if Wait_Cleanup'Count > 0 then
        Releasing_Cleanup := True;
    else
        Releasing_Cleanup := False;
        Final_Result := Commit;
    end if;
end Wait_Cleanup;
end Controller;

procedure T1(Params: Param) is
    X : Exception_Id;
    Decision : Vote_T;
begin
    select
        Controller.Wait_Abort(X); -- triggering event
        Raise_Exception(X); -- raise common exception
    then abort
        begin
            -- code to implement atomic action
            Controller.Done; --signal completion
        exception
            when E: others =>
                Controller.Signal_Abort(Exception_Identity(E));
        end;
    end select;
exception
    -- if any exception is raised during the action
    -- all tasks must participate in the recovery
    when E: others =>
        -- Exception_Identity(E) has been raised in all tasks

```



```

-- handle exception
if Handled_Ok then
    Controller.Cleanup(Commit);
else
    Controller.Cleanup(Aborted);
end if;
Controller.Wait_Cleanup(Decision);
if Decision = Aborted then
    raise Atomic_Action_Failure;
end if;
end T1;

procedure T2(Params : Param) is ...;

procedure T3(Params : Param) is ...;
end Action;

```

Each component of the action (T1, T2 and T3) has identical structure. The component executes a select statement with an abortable part. The triggering event is signalled by the Controller protected object if any component indicates that an exception has been raised and not handled locally in one of the components. The abortable part contains the actual code of the component. If this code executes without incident, the Controller is informed that this component is ready to commit the action. If any exceptions are raised during the abortable part, the Controller is informed and the identity of the exception passed. Note that, unlike backward error recovery (given earlier), here the cause of the error must be communicated.

If the Controller has received notification of an unhandled exception, it releases all tasks waiting on the Wait_Abort triggering event (any task late in arriving will receive the event immediately it tries to enter into its select statement). The tasks have their abortable parts aborted (if started), and the exception is raised in each task by the statement after the entry call to the controller. If the exception is successfully handled by the component, the task indicates that it is prepared to commit the action. If not, then it indicates that the action must be aborted. If any task indicates that the action is to be aborted, then all tasks will raise the exception Atomic_Action_Failure. Figure 7.6 shows the approach using a simple state transition diagram.

The above example illustrates that it is possible to program atomic actions with forward error recovery in Ada. However, there are two points to note about this example.

- Only the first exception to be passed to the Controller will be raised in all tasks. It is not possible to get concurrent raising of exceptions, as any exception raised in an abortable part when it is aborted is lost.
- The approach does not deal with the deserter problem. If one of the participants in the action does not arrive, the others are left waiting at the end of the action. To cope with this situation, it is necessary for each task to log its arrival with the action controller (see Exercise 7.7).

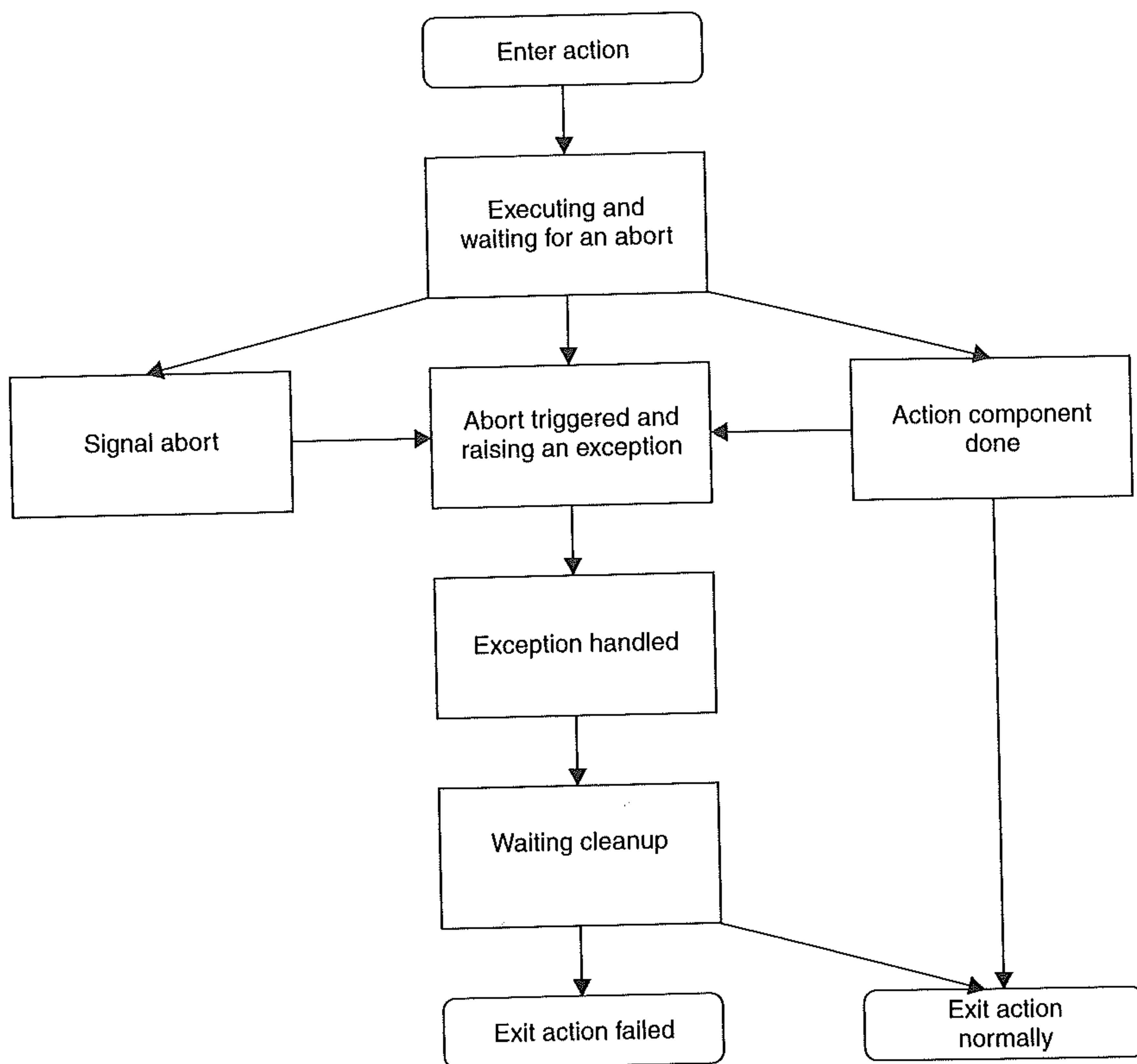


Figure 7.6 Simple state transition diagram illustrating forward error recovery.

7.7 Asynchronous notification in Real-Time Java

Real-Time Java supports both the resumption and termination models of asynchronous notification, but provides no direct support for asynchronous thread termination. The resumption model is provided by an asynchronous event-handling facility, where the handlers are scheduled entities rather than executed in the context of an interrupted thread. The termination model is provided within the context of an asynchronous exception-handling model for real-time threads. The latter can be used to program safe asynchronous termination. Note that facilities discussed apply only to Java systems that support the Real-Time Specification for Java; they cannot be used in a normal Java environment.

7.7.1 Asynchronous event handling

The equivalent of a C/Real-Time POSIX signal in Real-Time Java is an asynchronous event. Indeed, there is even a class `POSIXSignalHandler` which allows POSIX signals to be mapped onto Real-Time Java events for the occasion when Real-Time Java is being implemented on top of a POSIX-compliant operating system (see Section 14.4.2).

Program 7.4 shows the three main classes associated with asynchronous events in Real-Time Java. Each `AsyncEvent` can have one or more `AsyncEventHandlers`.

Program 7.4 The AsyncEvent, AsyncEventHandler and BoundAsyncEventHandler classes.

```
public class AsyncEvent {
    public AsyncEvent();

    public void addHandler(AsyncEventHandler handler);
    public void removeHandler(AsyncEventHandler handler);
    public void setHandler(AsyncEventHandler handler);
    // Associate a new handler with this event,
    // removing all existing handlers.

    public void bindTo(java.lang.String happening);
    // bind to external event
    public void unbindTo(java.lang.String happening);

    public ReleaseParameters createReleaseParameters();
    // creates a ReleaseParameters object representing the
    // characteristics of this event

    public void fire();
    // Schedule for execution the set of handlers for this event.
    public boolean handledBy(AsyncEventHandler handler);
    // Returns true if this event is handled by this handler.
}

public abstract class AsyncEventHandler implements Schedulable {
    public AsyncEventHandler();
    // parameters are inherited from the current thread

    public AsyncEventHandler(SchedulingParameters scheduling,
        ReleaseParameters release, MemoryParameters memory,
        MemoryArea area, ProcessingGroupParameters group);
    ... // other constructors available

    // methods which implement the Schedulable interface,
    // see Chapter 12

    protected int getAndClearPendingFireCount();
    // Atomically set to zero the number of pending executions
    // of this handler and returns the value from before
    // it was cleared.
    protected int getAndDecrementPendingFireCount();
    protected int getAndIncrementPendingFireCount();

    public void handleAsyncEvent();
    // Override this method to define the action to be
    // taken by this handler

    public final void run();
}

public abstract class BoundAsyncEventHandler extends AsyncEventHandler {
    public BoundAsyncEventHandler();
    // other constructors
}
```

When the event occurs (indicated by a call to the `fire` method), all the handlers associated with the event are scheduled for execution according to their `SchedulingParameters` – see Section 12.7. Note that the firing of an event can also be associated with the occurrence of an implementation-dependent external action by using the `bindTo` method.

Each handler is scheduled once for each outstanding event firing. Note, however, that the handler can modify the number of outstanding events by the methods in the `AsyncEventHandler` class.

Although an event handler is a schedulable entity, the goal is that it will not suffer the same overhead as an application thread. Consequently, it cannot be assumed that there is a separate implementation thread for each handler, as more than one handler may be associated with a particular implementation thread. If a dedicated thread is required, the `BoundAsyncEventHandler` should be used.

7.7.2 Asynchronous transfer of control in Real-Time Java

Early versions of Java allowed one thread to asynchronously affect another thread; however, this support has now been deprecated. Standard Java now only supports the following:

```
public void interrupt() throws SecurityException;
public boolean isInterrupted();
```

One thread can signal an interrupt to another thread by calling the `interrupt` method. The result of this depends on the current status of the interrupted thread.

- If the interrupted thread is blocked in the `wait`, `sleep` or `join` methods, it is made runnable and the `InterruptedException` is thrown.
- If the interrupted thread is executing, a flag is set indicating that an interrupt is outstanding. *There is no immediate effect on the interrupted thread.* Instead, the called thread must periodically test to see if it has been ‘interrupted’ using the `isInterrupted` method.

By itself, this does not meet the user needs outlined in Section 7.4.1.

Real-Time Java provides an alternative approach for interrupting a thread, based on asynchronous transfer of control (ATC). The Real-Time Java ATC model is similar to the Ada one in that it is necessary to indicate which regions of code can receive the ATC request. However, the Real-Time Java model is different in two important respects.

- (1) The Real-Time Java model is integrated with the Java exception-handling facility, whereas the Ada model is integrated into the `select` statement and entry-handling mechanisms.
- (2) The Real-Time Java model requires each method to indicate that it is prepared to allow the ATC to occur. ATC is deferred until the thread is executing within such a method. In contrast, Ada’s default is to allow the ATC if a subprogram has been called from within the `select-then-abort` statement; a deferred response must be explicitly handled.

Both languages defer the ATC during interaction with other threads/tasks (for example synchronized statements/methods in Java and protected actions and rendezvous in Ada) or within constructors and finalization (finally) clauses.

The Real-Time Java ATC model brings together the Java exception-handling model and an extension of thread interruption. Essentially the model is that when a real-time thread (or more generally a schedulable object – see Section 12.7) is interrupted, an asynchronous exception (`AsynchronouslyInterruptedException`) is delivered to the real-time thread rather than the thread having to poll for the interruption as would be the case with conventional Java. `AsynchronouslyInterruptedException` is a checked exception.

The notion of an asynchronous exception is not new and has been explored in previous languages. The main problem with them is how to program safely in their presence. As discussed in Chapter 3, most exception-handling mechanisms have exception propagation within a termination model. Consider a thread that has called method A, which has called method B, which has called method C. When an exception is raised within method C, if there is no local handler, the call to method C is terminated and a handler is sought in method B (the exception propagates up the call chain). If no handler is found in B, the exception is propagated to A. When a handler is found, it is executed, and the program continues to execute in the context in which the handler was found. There is no return to the context where the original exception was thrown. This model makes it difficult to write code that is tolerant of an asynchronous exception being thrown at it. Every method would need a handler for the root class of all asynchronous exceptions.

The Real-Time Java solution to this problem is to require that all methods (including constructors) that are prepared to allow the delivery of an asynchronous exception, place the exception in their `throws` lists; Real-Time Java calls such methods **AI-methods** (Asynchronously Interruptible). If a method does not do this, then the asynchronous exception is not delivered but held pending until the real-time thread is in a method that has the asynchronous exception in its throw clause. Hence, code that has been written without being concerned with ATC can execute safely even in an environment where ATCs are being used. Furthermore, to ensure that ATCs can be handled safely, Real-Time Java requires that:

- (1) ATCs are deferred during the execution of synchronized methods or statements and static initializers. This is to ensure that any shared data is left in a consistent state; Real-Time Java calls these sections of code and the methods that are not AI methods collectively **ATC-deferred** sections.
- (2) An ATC can only be handled from within code that is an ATC-deferred section; this is to avoid the handler for one ATC being interrupted by another ATC being delivered.

The full model is best explained in two stages. The first is the low-level support and overall approach; the second is the use of the high-level support to provide a structured means for handling ATC. Use of the basic ATC facilities requires three activities:

- (1) declaring an `AsynchronouslyInterruptedException` (AIE);
- (2) identifying methods which can be interrupted;
- (3) signalling an `AsynchronouslyInterruptedException` to a thread.

Program 7.5 The Real-Time Java Asynchronously InterruptedException class.

```

public class AsynchronouslyInterruptedException extends
    java.lang.InterruptedException
{
    public AsynchronouslyInterruptedException();

    public boolean clear();
    public boolean disable();
    // only valid within a doInterruptible,
    // returns true if successful
    public boolean doInterruptible (Interruptible logic);
    // Only one specific Interruptible can be running per thread
    // at any one time.
    // Returns True, if the Interruptible is executed, false if one
    // is already in progress for this thread.

    public boolean enable();
    public boolean fire();

    public static AsynchronouslyInterruptedException getGeneric();
    // returns the AsynchronouslyInterruptedException which
    // is generated when RealtimeThread.interrupt() is invoked

    public boolean isEnabled();
}

```

Program 7.5 shows the specification of the Asynchronously Interrupted-Exception class. The methods will be explained in due course, but for now all that is required is to know that for each thread there is an associated generic AIE.

An AIE may be placed in the throws list associated with a method. For example, consider the following class which provides an interruptible service using a package which declares non-interruptible services (that is, ones which do not have throws lists containing Asynchronously InterruptedExceptions).

```

import nonInterruptibleServices.*;

public class InterruptibleService {
    public AsynchronouslyInterruptedException stopNow =
        AsynchronouslyInterruptedException.getGeneric();

    public boolean service()
        throws AsynchronouslyInterruptedException {
        //code interspersed with calls to NonInterruptibleServices
    }
}

```


Now assume that a real-time thread `rtThread` has called an instance of this class to provide the service:

```
public InterruptibleService IS = new InterruptibleService();

// code of thread, rtThread
if (IS.service()) { ... } else { ... };
```

and that another real-time thread interrupts `rtThread`:

```
rtThread.interrupt();
```

The consequences of this call depend on the current state of `rtThread` when the call is made.

- If `rtThread` is executing within an ATC-deferred section – that is, executing within a synchronized method (or block), a static initializer or within a method that has no `AsynchronouslyInterruptedException` declared in its throws list (such as those in the package `nonInterruptibleServices`) – the AIE is marked as pending. The exception is delivered as soon as `rtThread` leaves the ATC-deferred region and is executing in a method with an `AsynchronouslyInterruptedException` declared in its throws list (such as the service method).
- If `rtThread` is executing within an AI-method (and it is not within a synchronized block), then the method's execution is interrupted, and control is transferred (propagated) up the call chain until it finds a try block in an ATC-deferred region that has a catch clause naming `AsynchronouslyInterruptedException` (or a parent class). Any synchronized methods or statements that are terminated by this propagation have their monitor locks released (and their finally clauses executed). The handler is then executed.
- If `rtThread` is blocked inside a `sleep`, `join`, `MemoryArea.join` (or `joinAndEnter`) or `waitForNextPeriodInterruptible` method called from within an AI-method, `rtThread` is rescheduled and the `AsynchronouslyInterruptedException` is delivered.
- If `rtThread` is blocked inside a `wait`, `sleep`, `join`, `MemoryArea.join` (or `joinAndEnter`) or `waitForNextPeriodInterruptible` method called from within an ATC-deferred region, `rtThread` is rescheduled and the `AsynchronouslyInterruptedException` is thrown as a synchronous exception (it is a subclass of the `InterruptedException`) and it is also marked as pending. Even if the synchronous exception is handled, the asynchronous exception is redelivered as soon as `rtThread` enters an AI-method.

Once an ATC has been delivered and control is passed to an appropriate exception handler, it is necessary to ascertain whether the caught ATC is the one expected by the interrupted thread. If it is, the exception can be handled. The `clear` method defined in the class `AsynchronouslyInterruptedException` is used for this purpose. The `clear` method simply resets the pending state if the AIE is current and returns

true, indicating that the current AIE is now no longer pending. If the flag is not reset, the AIE will then be redelivered when control next enters an AI-method.

```
import NonInterruptibleServices.*;
public class InterruptibleService {
    public AsynchronouslyInterruptedException stopNow =
        new AsynchronouslyInterruptedException();

    public boolean Service()
        throws AsynchronouslyInterruptedException {
        try {
            // code interdispersed with calls to
            // NonInterruptibleServices
        }
        catch AsynchronouslyInterruptedException AIE {
            if(stopNow.clear()) {
                //, handle the ATC
            } else {
                // Cleanup and leave the current AIE still pending.
            }
        }
    }
}
```

Here, when the AIE is thrown, control is passed to the catch clause at the end of the try block. A handler is found for `AsynchronouslyInterruptedException`s. In order to determine whether the current `AsynchronouslyInterruptedException` is `stopNow`, a call is made to the `clear` method. This returns true if `stopNow` is the current exception. If it is not the current exception then, the real-time thread does some cleanup routines before exiting the handler. In the latter case, the AIE is still pending.

The Interruptible interface

The above discussion illustrates the basic mechanisms provided by Real-Time Java for handling ATCs. To facilitate their structured use, the language also provides an interface called `Interruptible` – see Program 7.6.

An object which wishes to provide an interruptible method does so by implementing the `Interruptible` interface. The `run` method is the method that is interruptible; the `interruptAction` method is called by the system if the `run` method is interrupted.

Program 7.6 The Real-Time Java Interruptible interface.

```
public interface Interruptible {
    public void interruptAction (
        AsynchronouslyInterruptedException exception);

    public void run (AsynchronouslyInterruptedException exception)
        throws AsynchronouslyInterruptedException;
}
```

Once this interface is implemented, the implementation can be passed as a parameter to the `doInterruptible` method in the `AsynchronouslyInterruptedException` class. The method can then be interrupted by calling the `fire` method in the `AsynchronouslyInterruptedException` class. Further control over `AsynchronouslyInterruptedException` is given by the `disable`, `enable` and `isEnabled` methods. A disabled `AsynchronouslyInterruptedException` is deferred until it is enabled. An example of this latter use will be given in Section 7.7.3.

Note that only one `doInterruptible` method for a particular `AsynchronouslyInterruptedException` can be active at one time. If a call is outstanding, the method returns immediately with a false value.

Multiple `AsynchronouslyInterruptedExceptions`

Given that `AsynchronouslyInterruptedException` can be deferred, it is possible for multiple ATCs to be deferred. This can happen when the `run` method of one class (which implements the `Interruptible` interface) calls a `doInterruptible` on an AIE. The associated `run` method may also call another `doInterruptible`. Hence it is possible for a thread to be executing nested `doInterruptibles`. Consider the following example:

```
import javax.realtime.*;

public class NestedATC {
    AsynchronouslyInterruptedException AIE1 = new
        AsynchronouslyInterruptedException();
    AsynchronouslyInterruptedException AIE2 = new
        AsynchronouslyInterruptedException();
    AsynchronouslyInterruptedException AIE3 = new
        AsynchronouslyInterruptedException();

    public void method1() {
        // ATC-deferred region
    }

    public void method2() throws AsynchronouslyInterruptedException {
        AIE1.doInterruptible
            (new Interruptible()
             {
                 public void run(AsynchronouslyInterruptedException e)
                     throws AsynchronouslyInterruptedException
                 {
                     method1();
                 }
                 public void interruptAction(
                     AsynchronouslyInterruptedException e)
                 {
                     if(AIE1.clear()) {
                         // recovery here
                     } else {
                         // cleanup
                     }
                 }
             })
    }
}
```

```

        }
    }
};

}

public void method3() throws AsynchronouslyInterruptedException {
    AIE2.doInterruptible
    (new Interruptible()
    {
        public void run(AsynchronouslyInterruptedException e)
            throws AsynchronouslyInterruptedException {
            method2();
        }
        public void interruptAction(
            AsynchronouslyInterruptedException e){
            if(AIE2.clear()) {
                // recovery here
            } else {
                // cleanup
            }
        }
    }
    );
}

public void method4() throws AsynchronouslyInterruptedException {
    AIE3.doInterruptible
    (new Interruptible()
    {
        public void run(AsynchronouslyInterruptedException e)
            throws AsynchronouslyInterruptedException {
            method3();
        }
        public void interruptAction(
            AsynchronouslyInterruptedException e) {
            if(AIE3.clear()) {
                // recovery here
            } else {
                // cleanup
            }
        }
    }
    );
}
}

```

Now suppose that a thread *t* has created an instance of *NestedATC* and has called *method4*, which has called *method3*, which has called *method2*, which has called *method1* which is an ATC-deferred region. Assume that the thread is interrupted by a call to *AIE2.fire()*. This is held pending. If *AIE3* now comes in then *AIE2* is discarded as *AIE3* is at a higher level in the nesting. If *AIE1* comes in then *AIE1* is discarded (because it is at a lower level). Once *method1* has returned, the currently pending AIE is thrown.

7.7.3 Real-Time Java and atomic actions

This section illustrates how Real-Time Java's ATC facilities can be used to implement atomic actions with forward error recovery.

First, an `AtomicActionException` is defined along with an `AtomicActionFailure` exception.

```
import javax.realtime.AsynchronouslyInterruptedException;

public class AtomicActionException extends
    AsynchronouslyInterruptedException {
    public static Exception cause;
    public static boolean wasInterrupted;
}

public class AtomicActionFailure extends Exception;
```

Using a `ThreeWayRecoverableAtomicAction` similar to that defined earlier:

```
public interface ThreeWayRecoverableAtomicAction {
    public void role1() throws AtomicActionFailure;
    public void role2() throws AtomicActionFailure;
    public void role3() throws AtomicActionFailure;
}
```

a `RecoverableAction` class can be implemented with a similar structure to that given for Ada.

```
import javax.realtime.*;

public class RecoverableAction
    implements ThreeWayRecoverableAtomicAction {
    protected RecoverableController Control;
    private final boolean abort = false;
    private final boolean commit = true;

    private AtomicActionException aae1, aae2, aae3;

    public RecoverableAction() // constructor {
        Control = new RecoverableController();
        // for recovery
        aae1 = new AtomicActionException();
        aae2 = new AtomicActionException();
        aae3 = new AtomicActionException();
    }

    class RecoverableController {
        protected boolean firstHere, secondHere, thirdHere;
        protected int allDone;
        protected int toExit, needed;
        protected int numberOfParticipants;
        private boolean committed = commit;
    }
```

```

RecoverableController() {
    // for synchronization
    firstHere = false;
    secondHere = false;
    thirdHere = false;
    allDone = 0;
    numberOfParticipants = 3;
    toExit = numberOfParticipants;
    needed = numberOfParticipants;
}

synchronized void first() throws InterruptedException {
    while(firstHere) wait();
    firstHere = true;
}

synchronized void second() throws InterruptedException {
    while(secondHere) wait();
    secondHere = true;
}

synchronized void third() throws InterruptedException {
    while(thirdHere) wait();
    thirdHere = true;
}

synchronized void signalAbort(Exception e) {
    allDone = 0;
    AtomicActionException.cause = e;
    AtomicActionException.wasInterrupted = true;
    // raise an AsynchronouslyInterruptedException
    // in all participants
    aae1.fire();
    aae2.fire();
    aae3.fire();
}

private void reset() {
    firstHere = false;
    secondHere = false;
    thirdHere = false;
    allDone = 0;
    toExit = numberOfParticipants;
    needed = numberOfParticipants;
    notifyAll();
}

synchronized void done() throws InterruptedException {
    allDone++;
    if(allDone == needed) {
        notifyAll();
    } else while(allDone != needed) {
        wait();
        if(AtomicActionException.wasInterrupted) {
            allDone--;
            return;
        }
    }
}

```



```

    }
    toExit--;
    if(toExit == 0) {
        reset();
    }
}

synchronized void cleanup(boolean abort) {
    if(abort) { committed = false; };
}

synchronized boolean waitCleanup()
    throws InterruptedException {
    allDone++;
    if(allDone == needed) {
        notifyAll();
    } else while (allDone != needed) {
        wait();
    }
    toExit--;
    if(toExit == 0) {
        reset();
    }
    return committed;
}

}

public void role1() throws AtomicActionFailure,
    AsynchronouslyInterruptedException {

    boolean Ok;
    // entry protocol
    // no AIE until inside the atomic action
    boolean done = false;
    while(!done) {
        try {
            Control.first();
            done = true;
        } catch(InterruptedException e) {
            // ignore
        }
    }

    // the following defines an interruptible
    // section of code, and a routine to be called
    // if the code is interrupted
    Ok = aae1.doInterruptible
        (new Interruptible()
        {
            public void run(AsynchronouslyInterruptedException e)
                throws AsynchronouslyInterruptedException {
                try {
                    // perform action
                    // if necessary call e.disable() and e.enable() to
                    // defer AIE
                    Control.done();
                }
                catch(Exception x) {

```

```

        Control.signalAbort(x);
    }
}

    public void interruptAction(
        AsynchronouslyInterruptedException e) {
        // no action required
    }
}
);
if(!Ok) throw new AtomicActionFailure();

if(aae1.wasInterrupted) {
    try {
        // try to recover
        Control.cleanup(commit);
        if(Control.waitCleanup() != commit) {
            throw new AtomicActionFailure();
        };
    }
    catch(Exception x) {
        throw new AtomicActionFailure();
    }
};
}

public void role2() throws AtomicActionFailure,
    AsynchronouslyInterruptedException
{
    /* similar to role1 */;
}

public void role3() throws AtomicActionFailure,
    AsynchronouslyInterruptedException
{
    /* similar to role1 */;
}
}

```

Note that the absence of direct language support for asynchronous transfer of control makes the Java solution appear more complicated than its Ada counterpart. This is mainly due to the requirement to construct Runnable objects to pass to the `doInterruptible` method.

Summary

Reliable execution of tasks is essential if real-time embedded systems are to be used in critical applications. When tasks interact, it is necessary to constrain their intertask communication so that recovery procedures can be programmed, if required. Atomic actions have been discussed in this chapter as a mechanism by which programs, consisting of many tasks, can be structured to facilitate damage confinement and error recovery.

Actions are atomic if they can be considered, so far as other tasks are concerned, to be indivisible and instantaneous, such that the effects on the

system are as if they are interleaved as opposed to concurrent. An atomic action has well-defined boundaries and can be nested. Resources used in an atomic action are allocated during an initial *growing phase*, and released either as part of a subsequent *shrinking phase* or at the end of the action (if the action is to be recoverable).

The syntax of an atomic action can be expressed by an action statement. The following statement executed within task P_1 indicates that P_1 wishes to enter into an atomic action with P_2 and P_3 :

```
action A with (P2, P3) do
    -- sequence of statements
end A;
```

P_2 and P_3 must execute similar statements.

A *conversation* is an atomic action with backward error recovery facilities (in the form of recovery blocks).

```
action A with (P2, P3) do
    ensure <acceptance test>
    by
        -- primary module
    else by
        -- alternative module
    else error
end A;
```

On entry to the conversation, the state of the task is saved. While inside the conversation, a task is only allowed to communicate with other tasks active in the conversation and general resource managers. In order to leave the conversation, all tasks active in the conversation must have passed their acceptance test. If any task fails its acceptance test, all tasks have their state restored to that saved at the start of the conversation and they execute their alternative modules. Conversations can be nested and if all alternatives in an inner conversation fail then recovery must be performed at an outer level.

Forward error recovery via exception handlers can also be added to atomic actions. If an exception is raised by one task then all tasks active in the action must handle the exception.

```
action A with (P2, P3) do
    -- the action
exception
    when exception_a =>
        -- sequence of statements
    when others =>
        raise atomic_action_failure;
end A;
```

Two issues that must be addressed when using this approach are the resolution of concurrently raised exceptions and exceptions in internal actions.

Few mainstream languages or operating systems directly support the notion of an atomic action or a recoverable atomic action. However, most communication and synchronization primitives allow the isolation property of an action to be programmed. To implement a recoverable action requires an asynchronous notification mechanism. This can either have resumption semantics (in which case it is called an asynchronous event-handling mechanism) or it can have termination semantics (in which case it is called asynchronous transfer of control). C/Real-Time POSIX supports asynchronous events using signals and a thread-cancelling mechanism. A signal can be handled, blocked or ignored. Real-Time Java also supports asynchronous events.

Both Ada and Real-Time Java provide the termination model of asynchronous transfer of control. The Ada mechanism is built on top of the select statement. Real-Time Java's ATC, in contrast, is integrated into its exception and thread interrupt mechanisms. These termination approaches, in combination with exceptions, allow for an elegant implementation of a recoverable action.

Further reading

- Anderson, T. and Lee, P. A. (1990) *Fault Tolerance Principles and Practice*, 2nd ed. London: Prentice Hall.
- Burns, A. and Wellings, A. J. (2007) *Concurrent and Real-Time Programming in Ada*. Cambridge: Cambridge University Press.
- Lea, D. (2000) *Concurrent Programming in Java: Design Principles and Patterns*, 2nd ed. Harlow: Prentice Hall.
- Lynch, N. A. (ed.) (1993) *Atomic Transactions*, Morgan Kaufmann Series in Data Management Systems. San Mateo, California: Morgan Kaufmann.
- Northcutt, J. D. (1987) *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel*. Orlando: Academic Press.
- Shrivastava, S. K., Mancini, L. and Randell, B. (1987) *On the Duality of Fault Tolerant Structures*, Lecture Notes in Computer Science, Volume 309, pp. 19–37. Springer-Verlag.
- Wellings, A. J. (2004) *Concurrent and Real-Time Programming in Java*. Chichester: Wiley.

Exercises

- 7.1 Distinguish between an atomic *action* and an atomic *transaction*. What is the relationship between an atomic transaction and a conversation?
- 7.2 Rewrite the `Action_X` Ada package given in Section 7.6.3 so that it becomes a general-purpose package for controlling a three-task conversation. (Hint: use generics.)
- 7.3 Can your solution to Exercise 7.2 be extended to cope with an arbitrary number of tasks participating in the atomic action?

- 7.4 What would be the implications of extending Ada to enable one task to raise an exception in another?
- 7.5 Compare and contrast asynchronous notification and exception handling.
- 7.6 In Section 7.6.3, backward and forward error recovery is shown between three Ada tasks. Show how they can be combined into a single solution that gives both forward and backward error recovery between the same three tasks.
- 7.7 Update the solution given in Section 7.6.1 to deal with the deserter problem.
- 7.8 Consider the following four fragments of code:

```
-- Fragment 1
select
  T.Call; -- an entry call to task T
  Flag := A;
or
  delay 10.0;
  Flag := B;
  -- code taking 2 seconds to execute
end select;
```

```
-- Fragment 2
select
  T.Call; -- an entry call to task T
  Flag := A;
else
  delay 10.0;
  Flag := B;
  -- code taking 2 seconds to execute
end select;
```

```
-- Fragment 3
select
  T.Call; -- an entry call to task T
  Flag := A;
then abort
  delay 10.0;
  Flag := B;
  -- code taking 2 seconds to execute
end select;
```

```
-- Fragment 4
select
  delay 10.0;
  Flag := A;
then abort
  T.Call; -- an entry call to task T
  Flag := B;
  -- code taking 2 seconds to execute
end select;
```

A rendezvous with `T.Call` takes 5 seconds to execute. What is the value of the `Flag` variable after the execution of each of the four fragments in each of the following cases? You may assume a `Flag` assignment statement takes zero execution time.

- (1) T.Call is available when the select is executed.
- (2) T.Call is NOT available when the select is executed and does not become available in the next 14 seconds.
- (3) T.Call is NOT available when the select is executed, but does become available after 2 seconds.
- (4) T.Call is NOT available when the select is executed but does become available after 8 seconds.

7.9 Consider the following package specification which provides a procedure to search part of a large character array for a *unique* fixed-length string. The procedure returns the position of the start of the string if it is found.

```
package Search_Support is
  type Array_Bounds is range 1 .. 1_000_000_000;
  type Large_Array is array(Array_Bounds) of Character;
  type Pointer is access Large_Array;

  type Search_String is new String(1..10);

  procedure Search(Pt: Pointer;
    Lower, Upper: Array_Bounds;
    Looking_For : Search_String;
    Found : out Boolean;
    At_Location : out Array_Bounds);
end Search_Support;
```

Three tasks wish to perform a concurrent search of the array for the same string; they are derived from a common task type.

```
task type Searcher(Search_Array: Pointer;
  Lower, Upper: Array_Bounds) is
  entry Find(Looking_For : Search_String);
  entry Get_Result(At_Location : out Array_Bounds);
end Searcher;
```

The string to be found is passed via an initial rendezvous with the tasks. Sketch the body of the task type (and any other objects you might need) so that when one task finds the string, all other tasks are *immediately* informed of the string's location so that further fruitless search is avoided. Assume that the SearchString will be found by one of the three tasks. Furthermore, all tasks must be prepared to pass back the result via the GetResult entry.

7.10 Consider the following Ada code fragment:

```
Error_1, Error_2 : exception;
task Watch;
task Signaller;

protected Atc is
  entry Go;
  procedure Signal;
private
```



```

    Flag : Boolean := False;
end Atc;

protected body Atc is
    entry Go when Flag is
    begin
        raise Error_1;
    end Go;

    procedure Signal is
    begin
        Flag := True;
    end Signal;
end Atc;

task body Watch is
begin
    ...
    select
        Atc.Go;
    then abort
        -- code taking 100 millisecond
        raise Error_2;
    end select;
    ...
exception
    when Error_1 =>
        Put_Line("Error_1 Caught");
    when Error_2 =>
        Put_Line("Error_2 Caught");
    when others =>
        Put_Line("Other Errors Caught");
end Watch;

task body Signaller is
begin
    ...
    Atc.Signal;
    ...
end Signaller;

```

Describe carefully the possible executions of this program fragment assuming that context switches between tasks can happen at any time.

- 7.11** A particular POSIX-based application consists of several periodic processes and has two modes of operation: MODE A and MODE B. The application has one process which only operates in MODE A. Sketch the design of this process assuming that when the system wishes to undertake a mode change, it sends a signal to all processes indicating the current mode of operation. Assume also the existence of a routine called WAITNEXTPERIOD, which will suspend the process until its next period of execution is due. Note that a mode change should only affect the process at the beginning of each period.
- 7.12** Illustrate how Ada's OOP model can be used to produce extensible atomic actions.

- 7.13 Compare and contrast the Ada and Java models for asynchronous transfer of control.
- 7.14 To what extent can standard Java be used to implement atomic actions?
- 7.15 Why have the Java routines `resume()`, `stop()` and `suspend()` been made obsolete?
- 7.16 Redo Exercise 7.11 for Real-Time Java.
- 7.17 Show how Ada's termination model of exception handling can be implemented in response to the receipt of a POSIX signal.

Chapter 8

Resource control

8.1	Resource control and atomic actions	8.6	Resource usage
8.2	Resource management	8.7	Deadlock
8.3	Expressive power and ease of use		Summary
8.4	The requeue facility		Further reading
8.5	Asymmetric naming and security		Exercises

Chapter 7 considered the problem of achieving reliable task cooperation. It was pointed out that coordination between tasks is also required if they are to share access to scarce resources such as external devices, files, shared data fields, buffers and encoded algorithms. These tasks were termed **competing** tasks. Much of the logical (that is, non-temporal) behaviour of real-time software is concerned with the allocation of resources between competing tasks. Although the tasks do not communicate directly with each other to pass information concerning their own activities, they may communicate to coordinate access to the shared resources. A few resources are amenable to unlimited concurrent access; however, most are in some way restricted in their usage.

As noted in Chapter 4, the implementation of resource entities requires some form of control agent. If the control agent is passive then the resource is said to be **protected** (or **synchronized**). Alternatively, if an active agent is required to program the correct level of control, the resource controller is called a **server**.

This chapter discusses the problem of reliable resource control. The general allocation of resources between competing tasks is considered. Although such tasks are independent of each other, the act of resource allocation has implications for reliability. In particular, failure of a task could result in an allocated resource becoming unavailable to other tasks. Tasks may be starved of resources if other tasks are allowed to monopolize them. Furthermore, tasks can become deadlocked by holding resources that other tasks require while at the same time requesting more resources.

8.1 Resource control and atomic actions

Although tasks need to communicate and synchronize to perform resource allocation, this need not be in the form of an atomic action. This is because the only information exchanged is that necessary to achieve harmonious resource sharing; it is not possible to exchange arbitrary information. As a result of this, the resource controller, be it in the form of a protected resource or a server, is able to ensure the global acceptability of any change to its local data. If this were *not* the case, then when a task (which had been allocated a resource) failed, it would be necessary to inform all tasks, which had recently communicated with the resource controller, of its failure. However, the code necessary for a particular task to communicate with the controller should be an atomic action; thus no other task in the system can disrupt the task when it is being allocated, or is freeing, a resource. Furthermore, a resource manager and client task may use forward and backward error recovery to cope with any anticipated or unanticipated error conditions.

Although in the last chapter it was shown that in general no real-time language directly supported atomic actions, the indivisibility effect can be achieved by careful use of the available communication and synchronization primitives.

8.2 Resource management

Concerns of modularity (in particular information hiding) dictate that resources must be encapsulated and be accessed only through a high-level procedural interface; for example, in Ada, a package should, wherever possible, be used. Note in all of these examples, the details of the actual resource are omitted.

```
package Resources is
  type Resource is ...;

  type Resource_Control is synchronized interface;
  procedure Allocate(RC: in out Resource_Control;
                    R : out Resource) is abstract;
  procedure Free(RC: in out Resource_Control;
                R : in out Resource) is abstract;
end Resources;
```

The use of a synchronized interface allows both active and passive resource controllers to be represented. If the resource manager is a server, then the body of `Resource_Control` will contain a task (or an access object to a task type). A protected resource will use a protected object within the package body.

With monitor-based synchronization, such as C/Real-Time POSIX with condition variables and mutexes or Java's synchronized classes, protected resources are naturally encapsulated within a monitor. For example, in Java:

```
public class Resource {
  ...
}
```



```

public interface ResourceControl {
    public Resource allocate();
    public void free(Resource r);
}

public class ResourceManager implements ResourceControl {
    public synchronized Resource allocate() {
        ...
    }

    public synchronized void free(Resource r) {
        ...
    }
}

```

Other forms of synchronization, such as busy waiting and semaphores, do not give the appropriate level of encapsulation, and will therefore not be considered further in this chapter.

The next section is concerned with a discussion of the expressive power and ease of use (usability) of various approaches to resource management. After this discussion, a further section on security will look at how a resource controller can protect itself against misuse.

8.3 Expressive power and ease of use

Bloom in 1979 suggested criteria for evaluating synchronization primitives in the context of resource management. This analysis forms the basis of this section, which looks at the expressive power and ease of use of synchronization primitives for resource control. The primitives to be evaluated are monitors/synchronized methods (with their use of condition synchronization), servers (with a message-based interface) and protected resources (implemented as protected objects). The latter two both use guards for synchronization, and hence one aspect of this analysis is a comparison between **condition synchronization** and **avoidance synchronization**.

Bloom uses the term 'expressive power' to mean the ability of a language to express required constraints on synchronization. Ease of use of a synchronization primitive encompasses:

- the ease with which it expresses each of these synchronization constraints;
- the ease with which it allows the constraints to be combined to achieve more complex synchronization schemes.

In the context of resource control, the information needed to express these constraints can be categorized (following Bloom) as follows:

- the type of service request;
- the order in which requests arrive;

- the state of the server and any objects it manages;
- the parameters of a request.

Bloom's original set of constraints included 'the history of the object' (that is, the sequence of all previous service requests). It is assumed here that the state of the object can be extended to include whatever historical information is needed. An addition to the list is however made, as Bloom did not include:

- the priority of the client.

A full discussion of task priority is given in Chapter 11. For the purpose of this chapter, the priority of a task is taken to be a measure of the task's importance.

As indicated above, there are in general two linguistic approaches to constraining access to a service. The first is the **conditional wait**: all requests are accepted, but any task whose request cannot currently be met is suspended on a queue. The conventional monitor typifies this approach: a task whose request cannot be met is queued on a condition variable, and resumed when the request can be serviced. The second approach is **avoidance**: requests are not accepted unless they can be met. The conditions under which a request can safely be accepted are expressed as a guard on the action of acceptance.

Each of the five criteria, introduced above, for assessing synchronization approaches will now be considered.

8.3.1 Request type

Information about the type of operation requested can be used to give preference to one type of request over another (for example, read requests over write requests to a real-time database). With monitor-based synchronization and synchronized methods, the read and write operations could be programmed as distinct procedures, but the semantics of a monitor usually imply that outstanding calls on these monitor procedures are handled in an arbitrary, priority or FIFO way. It is not possible, therefore, to deal with read requests first; nor is it feasible to know how many outstanding calls there are to monitor procedures.

In Ada, different request types can readily be represented by different entries in the server task or protected object. Before gaining access to the entity (in order to queue on an entry), there is again no way of being giving preference over other calls. However, a natural way of giving preference to particular requests once they are queued is through guards which use the 'count' attribute of entries. The following shows a case in which Update requests are intended to have priority over Modify requests:

```
protected Resource_Manager is
  entry Update(...);
  entry Modify(...);
  procedure Lock;
  procedure Unlock;
private
  Manager_Locked : Boolean := False;
  ...
end Resource_Manager;
```



```

protected body Resource_Manager is
  entry Update(...) when not Manager_Locked is
  begin
    ...
  end Update;

  entry Modify(...) when not Manager_Locked and
    Update'Count = 0 is
  begin
    ...
  end Modify;

  procedure Lock is
  begin
    Manager_Locked := True;
  end Lock;

  procedure Unlock is
  begin
    Manager_Locked := False;
  end Unlock;
end Resource_Manager;

```

With protected objects, only entries can have guards; procedures, once they gain access to the object, will execute immediately and hence preference to particular request types cannot be given using procedures.

8.3.2 Request order

Certain synchronization constraints may be formulated in terms of the order in which requests are received (to ensure fairness, for example, or to avoid starvation of a client). As has already been observed, monitors usually deal with requests in FIFO order, and therefore immediately meet this requirement. In Ada, outstanding requests of the same type (calls to the same entry) can also be serviced in a FIFO manner if the appropriate queuing policy is chosen. Outstanding requests of different types (for example, calls to different entries within a protected object) are, unfortunately, serviced in an arbitrary order with this policy. It is outside the programmer's control. Thus, there is no way of servicing requests of different types according to order of arrival unless a FIFO policy is used and all clients first call a common 'register' entry:

```

Server.Register;
Server.Action(...);

```

This double call is not, however, without difficulty (as will be explained in Section 8.3.4).

8.3.3 Server state

Some operations may be permissible only when the server and the objects it administers are in a particular state. For example, a resource can be allocated only if it is free, and an item can be placed in a buffer only if there is an empty slot. With avoidance

synchronization, constraints based on state are expressed as guards and, with servers, on the positioning of accept statements (or message receive operators). Monitors are similarly quite adequate – with condition variables being used to implement constraints.

8.3.4 Request parameters

The order of operations of a server may be constrained by information contained in the parameters of requests. Such information typically relates to the identity or (in the case of quantifiable resources like memory) to the size of the request. A straightforward monitor structure (in Java) for a general resource controller follows. A request for a set of resources contains a parameter that indicates the size of the set required; an array of resources is then returned if the request is successful. If not enough resources are available then the caller is suspended; when any resources are released, all suspended clients are woken up to see if their request can now be met.

```
public interface MultipleResourceControl {

    public Resource[] allocate(int size) throws
        IntegerConstraintError, InterruptedException;
    // see Section 3.3.2 for definition of IntegerConstraintError

    public void free(int size, Resource[] these);
}

public class MultipleResourceManager
    implements MultipleResourceControl {

    private final int maxResources = ...;
    private int resourcesFree;

    public MultipleResourceManager() {
        resourcesFree = maxResources;
    }

    public synchronized Resource[] allocate(int size) throws
        IntegerConstraintError, InterruptedException
    {
        if(size > maxResources) throw new
            IntegerConstraintError(1, maxResources, size);
        while(size > resourcesFree) wait();
        resourcesFree = resourcesFree - size;
        return ...;
    }

    public synchronized void free(int size, Resource[] these) {
        resourcesFree = resourcesFree + size;
        notifyAll();
    }
}
```

With simple avoidance synchronization, the guards only have access to variables local to the server (or protected object); the data being carried with the call cannot be accessed

until the call has been accepted. It is therefore necessary to construct a request as a double interaction. The following discusses how a resource allocator that will cater for this problem can be constructed as a server in Ada. The associated structures for a protected object (in Ada) that implements the same interface are left as exercises for the reader (see Exercises at the end of the chapter).

Resource allocation and Ada – an example

A way of tackling the problem in Ada is to associate an entry family with each type of request. Each permissible parameter value is mapped onto a unique index of the family so that requests with different parameters are directed to different entries. Obviously, this is only appropriate if the parameter is of discrete type.

The approach is illustrated in the following package, which shows how lack of expressive power leads to a complex program structure (that is, poor ease of use). Again, consider an example of resource allocation in which the size of the request is given as a parameter of the request. As indicated above, the standard approach is to map the request parameters onto the indices of a family of entries, so that requests of different sizes are directed to different entries. For small ranges, the technique described earlier for ‘request type’ can be used, with the select statements enumerating the individual entries of the family. However, for larger ranges a more complicated solution is needed.

```
package Multiple_Resources is
  type Resource is ...;

  type Resource_Array is array (Integer range <>) of Resource;

  type Multiple_Resource_Control is synchronized interface;
  procedure Allocate(Manager: in out Multiple_Resource_Control;
    Size : in Positive;
    R : out Resource_Array) is abstract;
  procedure Free(Manager: in out Multiple_Resource_Control;
    R : in out Resource_Array) is abstract;
end Multiple_Resources;

with Multiple_Resources; use Multiple_Resources;
package Multiple_Resources_Managers is
  Max_Resources : constant Integer := ...;
  subtype Resource_Range is Integer range 1..Max_Resources;

  Too_Many_Requested : exception;

  task type Multiple_Resources_Manager is
    new Multiple_Resource_Control with
      entry Sign_In(Size : Resource_Range);
      entry Allocate(Resource_Range); -- entry family
      overriding entry Free(R : in out Resource_Array);
  end Multiple_Resources_Manager;

  overriding procedure Allocate(
    Manager: in out Multiple_Resources_Manager;
    Size : Positive; R : out Resource_Array);
end Multiple_Resources_Managers;
```

```

package body Multiple_Resources_Managers is
  procedure Allocate(Manager: in out Multiple_Resources_Manager;
                     Size : Positive;
                     R : out Resource_Array) is

  begin
    if Size > Max_Resources then
      raise Too_Many_Requested;
    end if;
    Manager.Sign_In(Size);  -- size is a parameter
    Manager.Allocate(Size); -- size is an index into a family
    -- return resources
  end Allocate;

  task body Multiple_Resources_Manager is
    Pending : array(Resource_Range) of
      Natural := (others => 0);
    Resource_Free : Resource_Range := Max_Resources;
    Allocated : Boolean;
  begin
    loop
      select
        accept Sign_In(Size : Resource_Range) do
          Pending(Size) := Pending(Size) + 1;
        end Sign_In;
      or
        accept Free(R : in out Resource_Array) do
          Resource_Free := Resource_Free + R'Length;
          -- return resources to pool
        end Free;
      end select;
    loop -- main loop
      loop -- accept any pending Sign-In or Frees, do not wait
        select
          accept Sign_In(Size : Resource_Range) do
            Pending(Size) := Pending(Size) + 1;
          end Sign_In;
        or
          accept Free(R : in out Resource_Array) do
            Resource_Free := Resource_Free + R'Size;
          end Free;
        else
          exit;
        end select;
      end loop;
      Allocated := False;

      for Request in reverse Resource_Range loop
        if Pending(Request) > 0 and Resource_Free >= Request then
          accept Allocate(Request);
          Pending(Request) := Pending(Request) - 1;
          Resource_Free := Resource_Free - Request;
          Allocated := True;
          exit; -- loop to accept new Sign-Ins
        end if;
      end loop;
    end loop;
  end task body;
end Multiple_Resources_Managers;

```



```

        exit when not Allocated;
    end loop;
end loop;
end Multiple_Resources_Manager;
end Multiple_Resources_Managers;

```

The manager gives priority to large requests. In order to acquire resources, a two-stage interaction with the manager is required: a 'sign-in' request and an 'allocate' request. This double interaction is hidden from the user of the resource by encapsulating the manager in a package and providing a single procedure (`Allocate`) to handle the request.

When there are no outstanding calls, the manager waits for a sign-in request or a free request (to release resources). When a sign-in arrives, its size is noted in the array of pending requests. A loop is then entered to record all sign-in and free requests that may be outstanding. This loop terminates as soon as there are no more requests.

A for loop is then used to scan through the pending array, and the request with the greatest size that can be accommodated is accepted. The main loop is then repeated in case a greater size request has attempted to sign-in. If no request can be serviced, the main loop is exited and the manager waits for new requests.

The solution is complicated by the need for the double rendezvous transaction. It is also expensive, as an entry is required for every possible size.

A much simpler system is to be found in the language SR. Here guards are allowed to access (that is, refer to) 'in' parameters. A resource request is, therefore, only accepted when the server or protected resource knows that it is in a state in which the request can be accommodated. The double call is not needed. For example (using Ada-like code, but not valid Ada, for a protected resource):

```

protected Resource_Control is -- NOT VALID ADA
    entry Allocate(Size : Resource_Range);
    procedure Free(Size : Resource_Range);
private
    Resource_Free : Resource_Range := Max_Resources;
end Resource_Control;

protected body Resource_Control is

    entry Allocate(Size : Resource_Range)
        when Resources_Free >= Size is -- NOT VALID ADA
    begin
        Resource_Free := Resource_Free - Size;
    end Allocate;

    procedure Free(Size : Resource_Range) is
    begin
        Resource_Free := Resource_Free + Size;
    end Free;

end Resource_Control;

```

Although syntactically simple, this type of solution has the disadvantage that it is not clear under what circumstances the guards or barriers need to be re-evaluated. This can lead to inefficient implementations. An alternative approach is to keep the guards simple but to increase the expressive power of the communication mechanism by adding a **requeue** facility. This will be explained in detail in Section 8.4; however, in essence, it allows a call that has been accepted (that is, it has passed a guard evaluation) to be requeued on a new (or indeed the same) entry where it must again pass a guard. The following also motivates the need for a requeue facility.

Double interactions and atomic actions

In the above discussions, examples were given that require the client task to make a double call on the server. One of the main factors that necessitates this double interaction is the lack of expressive power in simple avoidance synchronization. To program reliable resource control procedures, this structure must, therefore, be implemented as an atomic action. Unfortunately, with Ada this guarantee cannot be made. Between the two calls, that is, after ‘sign-in’ but before ‘allocate’, an intermediate state of the client is observable from outside the ‘atomic action’:

```
begin
  Manager.Sign_In(Size);
  Manager.Allocate(Size);
end;
```

This state is observable in the sense that another task can abort the client between the two calls and leave the server in some difficulty.

- If the server assumes the client will make the second call, the abort will leave the server waiting for the call (that is, deadlocked).
- If the server protects itself against the abort of a client (by not waiting indefinitely for the second call), it may assume the client has been aborted when in fact it is merely slow in making the call; hence the client is blocked erroneously.

In the context of real-time software, three approaches have been advocated for dealing with the abort problem.

- Define the abort primitive to apply to an atomic action rather than a task; forward or backward error recovery can then be used when communicating with the server.
- Assume that abort is only used in extreme situations where the breaking of the atomic action is of no consequence.
- Try to protect the server from the effect of client abort.

The third approach, in Ada, involves removing the need for the double call by requeuing the first call (rather than have the client make the second call). As indicated above, this will be explained in more detail in Section 8.4 after the final assessment criterion has been dealt with.

8.3.5 Requester priority

The final criterion for evaluating synchronization primitives for resource management involves the use of client priority. If a collection of tasks is runnable then the dispatcher can order their executions according to priority. The dispatcher cannot, however, have any control over tasks suspended waiting for resources. It is therefore necessary for the order of operations of the resource manager also to be constrained by the relative priorities of the client tasks.

In Ada, Real-Time Java and C/Real-Time POSIX it is possible to define a queuing policy that is priority ordered but, in general concurrent programming languages, tasks are released from primitives such as semaphores or condition variables in either an arbitrary or FIFO manner; monitors are often FIFO on entry; selective waits are often arbitrary or have a static textual priority ordering. In this latter case, it is possible to program clients so that they access the resource via a different interface. For a small priority range, this is now equivalent to the *Request Type* constraint. For large priority ranges, it becomes equal to using *Request Parameters*, and hence the earlier approaches can be used.

Although monitors are often described as having a FIFO queue discipline, this is not really a fundamental property. Priority ordered monitors are clearly possible. Indeed, the C/Real-Time POSIX and Real-Time Java implementations of monitors not only allow priority queues but also (conceptually) merge the external queue (of tasks waiting to enter the monitor) and the internal one (of tasks released by the signalling of a condition variable) to give a single priority-ordered queue. Hence, a higher-priority task waiting to gain access to the monitor will be given preference over a task that is released internally.

8.3.6 Summary so far

In the above discussions, five requirements have been used to judge the appropriateness of current language structures for dealing with general resource control. Monitors, with their condition synchronization, deal well with request parameters; avoidance-based primitives needed to be augmented to deal with parameters but have the edge on request type.

It should, however, be noted that the requirements themselves are not mutually consistent. There may well be conflict between the priority of the client and the order of arrival of the requests; or between the operation requested and the priority of the requester.

It has been shown that it is necessary to construct the client's interaction with a resource manager as an atomic action. The absence of direct language primitives to support atomic actions and the existence of abort and asynchronous transfer of control (ATC) make this difficult; tasks can be asynchronously removed from monitors or terminated between two related calls to a server task. Abort is used to eliminate erroneous or redundant tasks. If a monitor is constructed correctly then it can be assumed that a rogue task can do no damage while in a monitor (given that the knowledge that it is rogue was only available after it entered). Similarly a rogue client can do no damage if it is just waiting to make a second synchronous call.

It follows that there are well-specified situations in which a task should not be aborted (to be accurate, the effect of the abort will still occur but is postponed). This leads to the notion of **abort-deferred regions**.

Real-Time Java has the notion of an ATC-deferred region. However, the `destroy` method does cause problems for resource controllers, and therefore should not be used. Similar arguments apply to other obsolete methods in the `Java Thread` class.

Ada defines the execution of a protected object (and a rendezvous) to be an abort-deferred region. However, interactions with resource controllers are usually potentially suspending and such calls are not allowed from within a protected object. For example, a double call on a server task could *not* be encapsulated within a protected object. However, as indicated earlier, the `requeue` solves many of these resource control problems; it is described in the next section.

8.4 The requeue facility

The discussion of Bloom's criteria has shown that avoidance synchronization leads to a more structured way of programming resource managers, but lacks expressive power when compared with lower-level condition synchronization. One means of enhancing the usability of avoidance synchronization is to add a `requeue` facility. The Ada language has such a facility, and hence the following discussion will focus on that language's model.

The key notion behind `requeue` is to move the task (which has been through one guard or barrier) to 'beyond' another guard. For an analogy, consider a person (task) waiting to enter a room (protected object) which has one or more doors (guarded entries) giving access to the room. Once inside, the person can be ejected (requeued) from the room and once again be placed behind a (potentially closed) door.

Ada allows `requeues` between task entries and protected object entries. A `requeue` can be to the same entry, to another entry in the same unit, or to another unit altogether. `Requeues` from task entries to protected object entries (and vice versa) are allowed. However, the main use of `requeue` is to send the calling task to a different entry of the same unit from which the `requeue` was executed.

In Section 7.6.3, code was given that implemented forward error recovery and used a double interaction with `Controller` (that is, a call to `Cleanup` followed by a call to `Wait_Cleanup`). This could have been coded as a single call on `Cleanup` by `requeuing` onto `Wait_Cleanup` from within the protected entry (the `Wait_Cleanup` would then become a private entry):

```
entry Cleanup (Vote: Vote_T; Result : out Vote_T) when True is
begin
  if Vote = Aborted then
    Final_Result := Aborted;
  end if;
  requeue Wait_Cleanup with abort;
end Cleanup;
```

The 'with abort' facility will be described later. The effect of the call is to move the calling task onto the `Wait_Cleanup` entry (just as if the call had been made from outside).

The execution of the requeue statement terminates the execution of the originally called entry.

The resource control problem provides another illustrative example of the application of requeue. In the following algorithm, an unsuccessful request is requeued on to a private entry (called Assign) of the protected object. The caller of this protected object now makes a single call on Allocate. Whenever resources are released, a note is taken of how many tasks are on the Assign entry. This number of tasks can then retry and either obtain their allocations or be requeued back onto the same Assign entry. The last task to retry opens the barrier:

```

with Multiple_Resources; use Multiple_Resources;
package Resource_Controllers is

    Max_Resources : constant Integer := ...;
    subtype Resource_Range is Integer range 1..Max_Resources;
    protected type Resource_Controller is new
        Multiple_Resource_Control with
            overriding entry Allocate(Size : in Positive;
                                     R : out Resource_Array);
            overriding procedure Free(R : in out Resource_Array);
    private
        entry Assign(Size : Positive; R : out Resource_Array);
        Available : Resource_Range := Resource_Range'Last;
        New_Resources_Released : Boolean := False;
        To_Try : Natural := 0;
        ...
    end Resource_Controller;
end Resource_controllers;

package body Resource_Controllers is
    protected body Resource_Controller is
        entry Allocate(Size : Positive; R : out Resource_Array)
            when Available > 0 is
            begin
                if Size <= Available then
                    Available := Available - Size;
                    -- allocate
                else requeue Assign;
                end if;
            end Allocate;

        entry Assign(Size : Positive; R : out Resource_Array)
            when New_Resources_Released is
            begin
                To_Try := To_Try - 1;
                if To_Try = 0 then
                    New_Resources_Released := False;
                end if;
                if Size <= Available then
                    Available := Available - Size;
                    -- allocate
                else requeue Assign;
                end if;
            end Assign;
    end Resource_Controller;
end Resource_Controllers;

```

```

procedure Free(R : in out Resource_Array) is
begin
    Available := Available + R'Last;
    -- free resources
    if Assign'Count > 0 then
        -- there are unallocated requests outstanding
        To_Try := Assign'Count;
        New_Resources_Released := True;
    end if;
    end Free;
end Resource_Controller;
end Resource_Controllers;

```

Note that this will only work if the Assign entry queuing discipline is FIFO. It is left as an exercise for the reader to develop a priority-based algorithm.

It should be observed that a more efficient algorithm can be derived if the protected object records the smallest outstanding request. The barrier should then only be opened in Free (or remain opened in Assign) if Available \geq Smallest.

Even with this style of solution, however, it is difficult to give priority to certain requests other than in FIFO or task priority order. As indicated earlier, to program this level of control requires a family of entries. However, with requeue, an improved structure can be given; that is rather than

```

procedure Allocate(Size : Resource_Range) is
begin
    Manager.Sign_In(Size); -- size is a parameter
    Manager.Allocate(Size); -- size is a family index
end Allocate;

```

with the server task having:

```

accept Sign_In(Size : Resource_Range) do
    Pending(Size) := Pending(Size) + 1;
end Sign_In;

```

the double call can be made atomic by using:

```

procedure Allocate(Size : Resource_Range) is
begin
    Manager.Sign_In(Size); -- size is a parameter
end Allocate;

```

and

```

accept Sign_In(Size : Resource_Range) do
    Pending(Size) := Pending(Size) + 1;
    requeue Allocate(Size);
end Sign_In;

```


The server task can then be made more secure by having its `Allocate` entry private.

```
task type Multiple_Resources_Manager is new Multiple_Resource_Control with
  entry Sign_In(Size : Resource_Range);
  overriding entry Free(R : in out Resource_Array);
private
  entry Allocate(Resource_Range); -- entry family
end Multiple_Resources_Manager;
```

This algorithm is not only more straightforward than the one given earlier, but it also has the advantage that it is resilient to a task removing itself from an entry queue (after the count attribute has acknowledged its presence). Once a task has been requeued it cannot be aborted or subject to a time out on the entry call – see following discussion.

8.4.1 Semantics of requeue

It is important to appreciate that requeue is not a simple call. If procedure P calls procedure Q, then, after Q has finished, control is passed back to P. But if entry X requeues on entry Y, control is not passed back to X. After Y has completed, control passes back to the object that called X. Hence, when an entry or accept body executes a requeue, that body is completed.

One consequence of this is that when a requeue is from one protected object to another then mutual exclusion on the original object is given up once the task is queued. Other tasks waiting to enter the first object will be able to do so. However, a requeue to the same protected object will retain the mutual exclusion lock (if the target entry is open).

The entry named in a requeue statement (called the **target** entry) must either have no parameters or have a parameter profile that is equivalent (that is, type conformant) with that of the entry (or accept) statement from which the requeue is made. For example, in the resource control program, the parameters of `Assign` are identical to those of `Allocate`. Because of this rule, it is not necessary to give the actual parameters with the call; indeed, it is forbidden to do so (in case the programmer tries to change them). Hence if the target entry has no parameters, no information is passed; if it has parameters, then the corresponding parameters in the entity that is executing the requeue are mapped across. Because of this rule, the algorithm given earlier to get FIFO ordering over a number of different entries – that is, using a single register entry:

```
Server.Register;
Server.Action(...);
```

– cannot be programmed as a single call using requeue (as the parameters to all the different actions may not be identical).

An optional ‘with abort’ clause can be used with the requeue statement. Usually when a task is on an entry queue, it will remain there until serviced unless it made a timed entry call (see Section 9.4.2) or is removed due to the use of asynchronous transfer of control or abort. Once the task has been accepted (or starts to execute the entry of a protected object), the timeout is cancelled and the effect of any abort attempt is postponed

until the task comes out of the entry. There is, however, a question as to what should happen with requeue. Consider the abort issue; clearly two views can be taken.

- As the first call has been accepted, the abort should now remain postponed so that the task or protected object can be assured that the second call is there.
- If the requeue puts the calling task back onto an entry queue, then abort should again be possible.

A similar argument can be made with timeouts. The requeue statement allows both views to be programmed; the default does not allow further timeouts or aborts, the addition of the 'with abort' clause enables the task to be removed from the second entry. These semantics are needed, for example, with the forward error recovery algorithm given at the beginning of this section.

The real issue (in deciding whether to use 'with abort' or not) is whether the protected object (or server task) having requeued the client task expects it to be there when the guard/barrier is opened. If the correct behaviour of the object requires the task's presence, then 'with abort' should *not* be used.

8.4.2 Requeuing to other entries

Although requeuing to the same entity represents the normal use of requeue, there are situations in which the full generality of this language feature is useful.

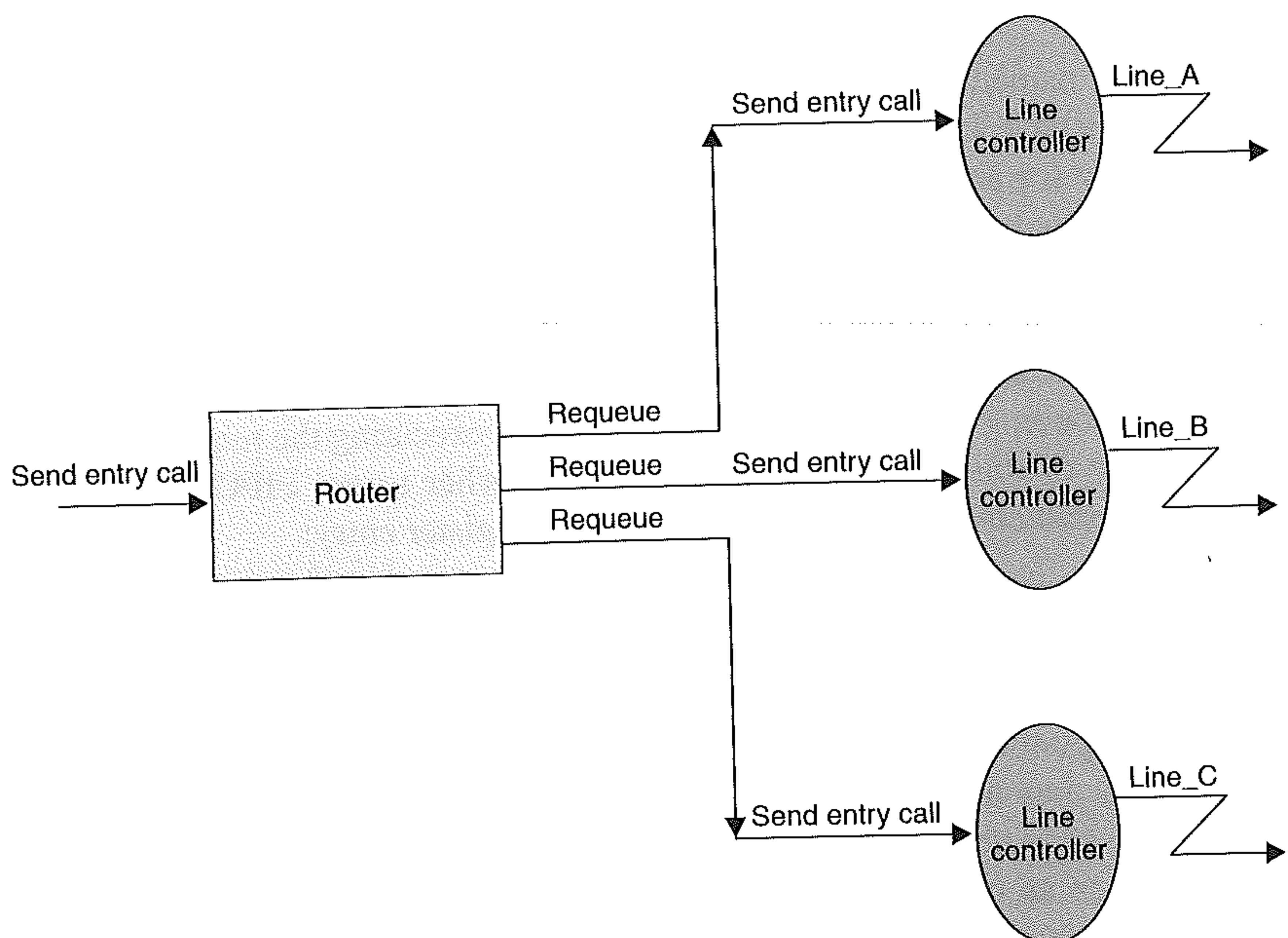


Figure 8.1 A network router.

Consider the situation in which resources are controlled by a hierarchy of objects. For example, a network router might have a choice of three communication lines on which to forward messages: `Line_A` is the preferred route, but if it becomes overloaded `Line_B` can be used; if this also becomes overloaded `Line_C` can be used. Each line is controlled by a server task; it is an active entity as it has housekeeping operations to perform. A protected unit acts as an interface to the router: it decides which of the three channels should be used and then uses requeue to pass the request to the appropriate server. The structure of the solution is illustrated in Figure 8.1, and the program is given below.

```

type Line_Id is (Line_A, Line_B, Line_C);
type Line_Status is array (Line_Id) of Boolean;

task type Line_Controller(Id : Line_Id) is
  entry Request(...);
end Line_Controller;

protected Router is
  entry Send(...); -- same parameter profile as Request
  procedure Overloaded(Line : Line_Id);
  procedure Clear(Line : Line_Id);
private
  Ok : Line_Status := (others => True);
end Router;

La : Line_Controller(Line_A);
Lb : Line_Controller(Line_B);
Lc : Line_Controller(Line_C);

task body Line_Controller is
  ...
begin
  loop
    select
      accept Request(...) do
        -- service request
      end Request;
    or
      terminate;
    end select;
    -- housekeeping including possibly
    Router.Overloaded(Id);
    -- or
    Router.Clear(Id);
  end loop;
end Line_Controller;

protected body Router is

  entry Send(...) when Ok(Line_A) or Ok(Line_B) or Ok(Line_C) is
  begin
    if Ok(Line_A) then
      requeue La.Request with abort;

```

```

    elsif Ok(Line_B) then
        requeue Lb.Request with abort;
    else
        requeue Lc.Request with abort;
    end if;
end Send;

procedure Overloaded(Line : Line_Id) is
begin
    Ok(Line) := False;
end Overloaded;

procedure Clear(Line : Line_Id) is
begin
    Ok(Line) := True;
end Clear;
end Router;

```

8.5 Asymmetric naming and security

In languages that have direct symmetric naming, a server task (or protected resource) always knows the identity of the client task with which it is dealing. This is also the case with indirect naming schemes based on a one-to-one intermediary. Asymmetric naming, however, results in the server being unaware of the identity of the client. It was noted earlier that this has the advantage that general-purpose servers can be written but it can lead to poor security on the use of resources. In particular, a server task may wish to know the identity of the calling client so that:

- a request can be refused on the grounds of deadlock prevention (see Section 8.7) or fairness (that is, to grant the request would be unfair to other clients);
- it can be guaranteed that resources are released only by the task that earlier obtained them.

In Ada it is possible for server to know the identity of its clients by using task identification (see Section 4.4) or in Java using `Thread.currentThread`. Consider the simple resource controller, in Ada, that only manages one resource.

```

protected Controller is
    entry Allocate;
    procedure Free;
private
    Allocated : Boolean := False;
    Current_Owner : Task_Id := Null_Task_Id;
end Controller;

protected body Controller is

    entry Allocate when not Allocated is
begin

```



```

    Allocated := True;
    Current_Owner := Allocated'Caller;
end Allocate;

procedure Free is
begin
    if Current_Task /= Current_Owner then
        raise Invalid_Caller; -- an appropriate exception
    end if;
    Allocated := False;
    Current_Owner := Null_Task_Id;
end Free;

end Controller;

```

Note that with this Ada facility, the caller of an entry is identified by the `Caller` attribute, whereas the caller of a procedure must be obtained from the function `Current_Task`. The reasons for this difference are not important here.

8.6 Resource usage

When competing or cooperating tasks require resources, the normal mode of operation is for them to: request the resource (waiting if necessary), use the resource, and then release it. A resource is normally requested for use in one of two modes of access. These are shared access or exclusive access. Shared access is where the resource can be used concurrently by more than one task: for example a read-only file. Exclusive access requires that only one task is allowed access to the resource at any one time: for example, a physical resource like a line printer. Some resources can be used in either mode. In this case, if a task requests access to the resource in sharable mode while it is being accessed in exclusive mode, then that task must wait. If the resource was already being accessed in sharable mode then the task can continue. Similarly, if exclusive access to a resource is requested then the task making the request must wait for the tasks currently accessing the resource in sharable mode to finish.

As tasks may be blocked when requesting resources, it is imperative that they do not request resources until they need them. Furthermore, once allocated they should release them as soon as possible. If this is not done then the performance of the system can drop dramatically as tasks continually wait for their share of a scarce resource. Unfortunately, if tasks release resources too soon and then fail, they may have passed on erroneous information through the resource. For this reason, the *two-phased* resource usage, introduced in Section 7.1.1, must be modified so that resources are not released until the action is completed. Any recovery procedure within the action must either release the resources, if successful recovery is performed, or undo any effects on the resource. The latter is required if recovery cannot be achieved and the system must be restored to a safe state. With forward error recovery, these operations can be performed by exception handlers. With backward error recovery, it is not possible to perform these operations without some additional language support.

8.7 Deadlock

With many tasks competing for a finite number of resources, a situation may occur where one task, T_1 , has sole access to a resource, R_1 , while waiting for access to another resource, R_2 . If task T_2 already has sole access to R_2 and is waiting for access to R_1 , then **deadlock** has occurred because both tasks are waiting for each other to release a resource. With deadlock, all affected tasks are suspended indefinitely. A similar acute condition is where a collection of tasks are inhibited from proceeding but are still executing. This situation is known as **livelock**. A typical example would be a collection of interacting tasks stuck in loops from which they cannot proceed but in which they are doing no useful work.

Another possible problem occurs when several tasks are continually attempting to gain sole access to the same resource; if the resource allocation policy is not fair, one task may never get its turn to access the resource. This is called **indefinite postponement** or **starvation** or **lockout** (see Chapter 5). In concurrent systems, **liveness** means that if something is supposed to happen, it eventually will. Breaches of liveness result from deadlock, livelock or starvation. Note that liveness is not as strong a condition as fairness. It is, however, difficult to give a single precise definition of fairness.

There are in general four necessary conditions that must hold if deadlock is to occur.

- **Mutual exclusion** – only one task can use a resource at once (that is, the resource is non-sharable or at least limited in its concurrent access).
- **Hold and wait** – there must exist tasks which are holding resources while waiting for others.
- **No preemption** – a resource can only be released voluntarily by a task.
- **Circular wait** – a circular chain of tasks must exist such that each task holds resources which are being requested by the next task in the chain.

If a real-time system is to be reliable, it must address the issue of deadlock. There are three possible approaches:

- deadlock prevention;
- deadlock avoidance;
- deadlock detection and recovery.

These strategies are not discussed further in this book as they are adequately dealt with in most Operating Systems textbooks. In Section 11.9.2 it will be shown how a particular method of scheduling tasks and resources (the Priority Ceiling Protocol) avoids deadlock.

Summary

In every computer system, there are many tasks competing for a limited set of resources. Algorithms are required which both manage the resource allocation/deallocation procedures (the mechanism of resource allocation) and which guarantee that resources are allocated to tasks according to a predefined

behaviour (the policy of resource allocation). These algorithms are also responsible for ensuring that tasks cannot deadlock while waiting for resource allocation requests to be fulfilled.

Sharing resources between tasks requires those tasks to communicate and synchronize. Therefore it is essential that the synchronization facilities provided by a real-time language have sufficient expressive power to allow a wide range of synchronization constraints to be specified. These constraints can be categorized as follows: resource scheduling must take account of:

- the type of service request;
- the order in which requests arrive;
- the state of the server and any objects it manages;
- the parameters of a request;
- the priority of the client.

Monitors (with condition synchronization) deal well with request parameters; avoidance synchronization in message-based servers or protected objects copes adequately with request types.

Where there is insufficient expressive power, tasks are often forced into a double interaction with a resource manager. This must be performed as an atomic action; otherwise it is possible for the client task to be aborted after the first interaction but before the second. If this possibility does exist then it is very difficult to program reliable resource managers. One means of extending the expressive power of guards is to allow requeueing. This facility enables avoidance synchronization to be as effective as condition synchronization.

Further reading

Bloom, T. (1979) Evaluating synchronization mechanisms. *Proceedings of the Seventh Symposium on Operating Systems Principles*, ACM, pp. 24–32.

Silberschatz, A., Galvin P. A. and Gagne, G. (2008) *Operating System Concepts*, 8th edn. New York: Wiley.

Exercises

8.1 The following resource controller attempts to associate priorities with request:

```
type Level is (Urgent, Medium, Low);

task Controller is
    entry Request(Level) (D:Data);
end Controller;

task body Controller is
    ...
```

```

begin
  loop
    ...
    select
      accept Request(Urgent)(D:Data) do
        ...
      end;
    or
      when Request(Urgent)'Count = 0 =>
        accept Request(Medium)(D:Data) do
          ...
        end;
    or
      when Request(Urgent)'Count = 0 and
        Request(Medium)'Count = 0 =>
        accept Request(Low)(D:Data) do
          ...
        end;
    end select
    ...
  end loop;
end Controller;

```

Explain this solution and indicate under what conditions it will fail. Why would it not be advisable to extend the above solution to cope with a numeric priority which falls in the range 0 to 1000? Sketch an alternative solution which will cope with larger priority ranges. You may assume that the calling tasks issue simple entry calls and are not aborted.

- 8.2 Show how the resource manager given in Section 8.3.1 can be programmed in Java (which does not have a requeue facility).
- 8.3 Show how the Ada resource manager given in Section 8.3.4 can be programmed using protected objects.
- 8.4 Show how the Ada resource manager given in Section 8.4 can be updated to allow high-priority clients to be given preference over low-priority clients.
- 8.5 Show how C/Real-Time POSIX mutexes and condition variables can be used to implement a resource controller where several identical resources can be allocated and freed. Clients can request and free one or more resources using the following interface:

```

typedef struct {
    ... /* fill in */
} resource;

void allocate(int size, resource *R);
void deallocate(int size, resource *R);

void initialize(resource *R);

```

- 8.6 Show how the network router example given in Section 8.4.2 can be programmed in Java.

Chapter 9

Real-time facilities

9.1	The notion of time	9.6	Temporal scopes
9.2	Access to a clock		Summary
9.3	Delaying a task		Further reading
9.4	Programming timeouts		Exercises
9.5	Specifying timing requirements		

In Chapter 1, it was noted that a language for programming embedded systems requires facilities for real-time control. Indeed, the term ‘real-time’ has been used as a synonym for this class of system. Given the importance of time in many embedded systems, it may appear strange that consideration of this topic has been postponed until Chapter 9. Facilities for real-time control are, however, generally built upon the concurrency model within the language, and it is therefore necessary to have covered this area first.

The introduction of the notion of time into a programming language can best be described in terms of three largely independent topics.

- (1) Interfacing with ‘Time’; for example, accessing clocks so that the passage of time can be measured, delaying tasks until some future time, and programming timeouts so that the non-occurrence of some event can be recognized and dealt with.
- (2) Representing timing requirements; for example, specifying rates of execution and deadlines.
- (3) Satisfying timing requirements.

This chapter is largely concerned with the first two of these topics, although it will commence with some discussion of the notion of time itself. Chapter 11 considers ways of implementing systems such that the worst-case temporal behaviour can be predicted, and hence timing requirements ratified.

9.1 The notion of time

Our everyday experiences are so intrinsically linked to notions of past, present and future that it is surprising that the question ‘What is time?’ is still largely unresolved.

Philosophers, mathematicians, physicists and, more recently, engineers have all studied 'time' in minute detail, but there is still no consensus on a definitive theory of time. As St Augustine stated:

What, then, is time? If no one asks me, I know what it is. If I wish to explain it to him who asks me, I do not know.

A key question in the philosophical discussions of time can be stated succinctly as 'do we exist in time, or is time part of our existence?' The two mainstream schools of thought are Reductionism and Platonism. All agree that human (and biological) history is made up of events, and that these events are ordered. Platonists believe that time is a fundamental property of nature; it is continuous, non-ending and non-beginning, 'and possesses these properties as a matter of necessity'. Our notion of time is derived from a mapping of historical events onto this external time reference.

Reductionists, by comparison, do without this external reference. Historical time, as made up of historical events, is the only meaningful notion of time. By assuming that certain events are 'regular' (for example, sunrise, winter solstice, vibrations of atoms, and so on), they can invent a useful time reference that enables the passage of time to be measured. But this time reference is a construction, not a given.

A consequence of the Reductionist view is that time cannot progress without change occurring. If the universe started with a 'Big Bang' then this represents the very first historical event and hence time itself started with 'space' at this first epoch. For Platonists, the Big Bang is just one event on an unbounded time line.

Over large distances, Einstein showed that relativity effects impinge not only on time directly but also on the temporal ordering of events. Within the special theory of relativity, the observer of events imposes a frame of references. One observer may place event A before event B; another observer (in a different frame of reference) may reverse the order. Due to such difficulties with temporal ordering, Einstein introduced **causal ordering**. Event A may cause event B if all possible observers see event A occurring first. Another way of defining such causality is to postulate the existence of a signal that travels, at a speed no greater than the speed of light, from event A to event B.

These different themes of time are well illustrated by the notion of simultaneous events. To Platonists, the events are simultaneous if they occur 'at the same time'. For Reductionists, simultaneous events 'happen together'. With Special Relativity, simultaneous events are those for which a causal relationship does not exist.

From a mathematical point of view, there are many different topologies for time. The most common one comes from equating the passage of time with a 'real' line. Hence time is linear, transitive, irreflexive and dense:

- linearity: $\forall x, y : x < y \text{ or } y < x \text{ or } x = y$
- transitivity: $\forall x, y, z : (x < y \text{ and } y < z) \Rightarrow x < z$
- irreflexivity: $\forall x : \text{not } (x < x)$
- density: $\forall x, y : x < y \Rightarrow \exists z : (x < z < y)$

The engineering perspective can largely ignore the philosopher's issue of time. An embedded real-time computer system needs to coordinate its execution with the 'time'

of its environment. The term 'real' is used to draw a distinction with the computer's time. It is real because it is external. Whether this external frame of reference is a Reductionist's construction or an approximative for the Platonists' 'absolute' time frame is not significant. Moreover, for most applications, relativistic effects can also be ignored. In terms of the mathematical topology of real-time systems, there are conflicting opinions as to whether time should be viewed as dense or discrete. Because computers work in discrete time, there is some advantage in constructing computational models based upon discrete time. The significant experience that other branches of engineering have in using, to good effect, dense time models argues the other way. Attempts to integrate both approaches has merely led to a third approach – hybrid systems.

If, by general consensus, a series of events is deemed to be regular then it is possible to define a standard measurement of time. Many such standards have existed in the past. Table 9.1 gives a brief description of some of the more significant ones. This description is taken from Hoogetboom and Halang (1992).

9.2 Access to a clock

If a program is going to interact in any meaningful way with the time frame of its environment then it must have access to some method of 'telling the time' or, at least, have some way of measuring the passage of time. This can be done in two distinct ways:

- by having direct access to the environment's time frame;
- by using an internal hardware clock that gives an adequate approximation to the passage of time in the environment.

The first approach is becoming more common and can be achieved in a number of ways. The simplest is for the environment to supply a regular interrupt that is clocked internally. Alternatively the system can be fitted with radio receivers and use one of the international time signals. Universal Coordinated Time (UTC) signals are broadcast by land-based radio stations (on shortwave frequencies) and satellites. For example, GPS (Global Positioning System) incorporates a time signal. Radio signals typically have an accuracy of 0.1–10 milliseconds; GPS provides a much better service with an accuracy of about 1 microsecond. If the system is linked to its environment via a telephone line or a network then these may also provide a time service (with an accuracy of perhaps a few milliseconds). With the Internet, NTP (Network Time Protocol) does indeed provide such a service.

Internal hardware clocks are devices that count the number of oscillations that occur in a quartz crystal. They typically divide this count by a fixed number and store the result in a register (counter) that the system's software can access. Inevitably this 'time' count is not always in perfect synchronization with the external time reference. The error is called **clock drift**. Clock drift may occur due, for example, to temperature changes within the system. A standard quartz crystal may drift by perhaps 10^{-6} seconds per second, i.e. one second in 11.6 days. A high-precision clock may have drift values of 10^{-7} or 10^{-8} .

In distributed systems there is not only the drift between internal and external time that must be allowed for but also the **skew** between any of the clocks within the systems.

Name	Description	Note
True solar day	Time between two successive culminations (highest point of the Sun)	Varies through the year by 15 minutes (approx.)
Temporal hour	One-twelfth part of the time between sunrise and sunset	Varies considerably through the year
Universal Time (UT0)	Mean solar time at Greenwich meridian	Defined in 1884
Second (1)	1/86 400 of a mean solar day	
Second (2)	1/31 566 925.9747 of the tropical year for 1900	Ephemeris Time defined in 1955
UT1	Correction to UT0 because of polar motion	
UT2	Correction of UT1 because of variation in the speed of rotation of the Earth	
Second (3)	Duration of 9 192 631 770 periods of the radiation corresponding to the transition between two hyperfine levels of the ground state of the Caesium 133 atom	Accuracy of current Caesium atomic clocks deemed to be one part in 10^{13} (that is, one clock error per 300 000 years)
International Atomic Time (IAT)	Based upon Caesium atomic clock	
Universal Coordinated Time (UTC)	An IAT clock synchronized to UT2 by the addition of occasional leap ticks	Maximum difference between UT2 (which is based on astronomical measurement) and UTC (which is based upon atomic measurement) is kept to below 0.5 seconds

Table 9.1 Time standards.

If each node in the distributed system has its own clock then they will not provide a perfect source of time. If a *global time* service is needed then some form of clock synchronization protocol is needed. Many such protocols exist; they differ in how they deal with node and network failure. Further details on these protocols can be found in the literature on distributed systems – see for example Coulouris et al. (2005).

From the programmer’s perspective, access to time can be provided either by a clock primitive in the language or via a device driver for the internal clock, external clock or radio receiver. The programming of device drivers is the topic of Chapter 14; the following subsections illustrate how Ada, Java and C/Real-Time POSIX provide clock abstractions. In general these languages are silent about how these abstractions should be interpreted in distributed systems.

9.2.1 The clock packages in Ada

Access to a clock in Ada is provided by a predefined (compulsory) library package called `Calendar` and an optional real-time facility. The `Calendar` package (see Program 9.1) implements an abstract data type for `Time`. It provides a function `Clock` for reading the time and various subprograms for converting between `Time` and humanly understandable units, such as `Years`, `Months`, `Days` and `Seconds`. The first three of these are given as integer subtypes. `Seconds` are, however, defined as a subtype of the primitive type `Duration`.

Program 9.1 The Ada `Calendar` package.

```
package Ada.Calendar is

  type Time is private;

  subtype Year_Number is Integer range 1901..2099;
  subtype Month_Number is Integer range 1..12;
  subtype Day_Number is Integer range 1..31;
  subtype Day_Duration is Duration range 0.0..86400.0;

  function Clock return Time;

  function Year(Date:Time) return Year_Number;
  function Month(Date:Time) return Month_Number;
  function Day(Date:Time) return Day_Number;
  function Seconds(Date:Time) return Day_Duration;

  procedure Split(Date:in Time; Year:out Year_Number;
                  Month:out Month_Number; Day:out Day_Number;
                  Seconds:out Day_Duration);

  function Time_Of(Year:Year_Number; Month:Month_Number;
                  Day:Day_Number; Seconds:Day_Duration := 0.0)
    return Time;

  function "+"(Left:Time;Right:Duration) return Time;
  function "+"(Left:Duration;Right:Time) return Time;
  function "-"(Left:Time;Right:Duration) return Time;
  function "-"(Left:Time;Right:Time) return Duration;

  function "<"(Left,Right:Time) return Boolean;
  function "<="(Left,Right:Time) return Boolean;
  function ">"(Left,Right:Time) return Boolean;
  function ">="(Left,Right:Time) return Boolean;

  Time_Error:exception;
  -- Time_Error is raised by Time_Of, Split, "+", and "-"

private
  -- implementation dependent
end Ada.Calendar;
```

Type `Duration` is a predefined fixed-point real that represents an interval of time (relative time). Both its accuracy and its range are implementation-dependent, although its range must be at least $-86\,400.0 \dots 86\,400.0$, which covers the number of seconds in a day. Its granularity must be no greater than 20 milliseconds. In essence, a value of type `Duration` should be interpreted as a value in seconds. Note that in addition to the above subprograms, the `Calendar` package defines arithmetic operators for combinations of `Duration` and `Time` parameters and comparative operations for `Time` values.

The code required to measure the time taken to perform a computation is quite straightforward. Note the use of the `-` operator, which takes two `Time` values but returns a value of type `Duration`.

```
declare
  Old_Time, New_Time : Time;
  Interval : Duration;
begin
  Old_Time := Clock;
  -- other computations
  New_Time := Clock;
  Interval := New_Time - Old_Time;
end;
```

In the latest version of Ada (Ada 2005) some additional packages have been provided that help with:

- constructing code that manipulates time values (including days that may have leap seconds included/excluded);
- formatting time values for input and output; and
- dealing with time zones.

The other language clock is provided by the optional package `Real_Time`. This has a similar form to `Calendar` but is intended to give a finer granularity. The constant `Time_Unit` is the smallest amount of time representable by the `Time` type. The value of `Tick` must be no greater than one millisecond; the range of `Time` (from the epoch that represents the program's start-up) must be at least fifty years.

As well as providing a finer granularity, the `Clock` of `Real_Time` is defined to be **monotonic**. The `Calendar` clock is intended to provide an abstraction for a 'wall clock' and is, therefore, subject to leap years, leap seconds and other adjustments. A monotonic clock has *no* such adjustments. The `Real_Time` package is outlined in Program 9.2.

In addition to these real-time clocks Ada also provides clocks that measure the execution time of task. These facilities are discussed in Chapter 12.

9.2.2 Clocks in Real-Time Java

Standard Java supports the notion of a wall clock, and thus has facilities similar to Ada. The current time can be found in Java by calling the static method `System.currentTimeMillis` in the package `java.lang`. This returns the number of milliseconds since midnight, January 1, 1970 GMT. The `Date` and `Calendar` classes in `java.util` use this method as a default when constructing date objects.

Program 9.2 The Ada Real_Time package.

```

package Ada.Real_Time is
  type Time is private;
  Time_First: constant Time;
  Time_Last: constant Time;
  Time_Unit: constant := -- implementation-defined-real-number;

  type Time_Span is private;
  Time_Span_First: constant Time_Span;
  Time_Span_Last: constant Time_Span;
  Time_Span_Zero: constant Time_Span;
  Time_Span_Unit: constant Time_Span;

  Tick: constant Time_Span;
  function Clock return Time;

  function "+" (Left: Time; Right: Time_Span) return Time;
  ...

  function "<" (Left, Right: Time) return Boolean;
  ...

  function "+" (Left, Right: Time_Span) return Time_Span;
  ...

  function "<" (Left, Right: Time_Span) return Boolean;
  ...

  function "abs"(Right : Time_Span) return Time_Span;

  function To_Duration (Ts : Time_Span) return Duration;
  function To_Time_Span (D : Duration) return Time_Span;

  function Nanoseconds (Ns: Integer) return Time_Span;
  function Microseconds (Us: Integer) return Time_Span;
  function Milliseconds (Ms: Integer) return Time_Span;

  type Seconds_Count is range -- implementation-defined

  procedure Split(T : in Time; Sc: out Seconds_Count;
                  Ts : out Time_Span);
  function Time_Of(Sc: Seconds_Count; Ts: Time_Span) return Time;

private
  -- not specified by the language
end Ada.Real_Time;

```

Real-Time Java adds to these facilities real-time clocks with high-resolution time types. Program 9.3 shows a summary of the base definition of the high-resolution time class. There are methods to read, write and compare time values. This abstract class has two subclasses: one which represents absolute time and one which represents relative time (similar to Ada's *Duration* type). These are given in Program 9.4. Absolute time is actually expressed as a time relative to 1 January 1970, GMT.

Program 9.3 An extract of the Real-Time Java `HighResolutionTime` class.

```

public abstract class HighResolutionTime
    implements Comparable, Cloneable {
    // methods
    public int compareTo(HighResolutionTime time);
    public boolean equals(HighResolutionTime time);

    public final long getMilliseconds();
    public final int getNanoseconds();

    public void set(HighResolutionTime time);
    public void set(long millis);
    public void set(long millis, int nanos);

    public static void waitForObject(Object target,
        HighResolutionTime time) throws InterruptedException;
    ...
}

```

The `HighResolutionTime`, `AbsoluteTime` and `RelativeTime` classes are all reasonably self-explanatory. However, the method `absolute` does need further discussion. Its role is to convert the encapsulated time (be it absolute or relative) to an absolute time relative to some clock. If an `AbsoluteTime` object is passed as a parameter, the object is updated to reflect the encapsulated time value. Furthermore, the same object is also returned by the function. This is because if a null object is passed as a parameter, a new object is created and returned. In Java, the parameter is copied by value and, therefore, cannot be updated. Consequently, it is necessary for the function to create a new `AbsoluteTime` object and return it.

The Real-Time Java `Clock` class, given in Program 9.5, defines the abstract class from which all clocks are derived. The language allows many different types of clocks; for example, there could be an execution-time clock which measures the amount of execution time being consumed. There is always one real-time clock which advances in sync with the external world. The method `getRealtimeClock` allows this clock to be obtained.¹ Other methods are provided to get the resolution of a clock and, if the hardware permits, to set the resolution of a clock.

The code to measure the time taken to perform a computation is:

```

{
    AbsoluteTime oldTime, newTime;
    RelativeTime interval;
    Clock clock = Clock.getRealtimeClock();

    oldTime = clock.getTime();
    // other computations
    newTime = clock.getTime();

    interval = newTime.subtract(oldTime);
}

```

¹Note that this is a static method, and therefore it can be called directly without knowledge of any subclasses.

Program 9.4 The Real-Time Java Absolute and Relative classes.

```

public class AbsoluteTime extends HighResolutionTime {
    // various constructor methods including
    public AbsoluteTime(AbsoluteTime T);
    public AbsoluteTime(long millis, int nanos);

    public AbsoluteTime absolute(Clock clock, AbsoluteTime dest);

    public AbsoluteTime add(long millis, int nanos);
    public final AbsoluteTime add(RelativeTime time);
    ...
    public final RelativeTime subtract(AbsoluteTime time);
    public final AbsoluteTime subtract(RelativeTime time);
}

public class RelativeTime extends HighResolutionTime {
    // various constructor methods including
    public RelativeTime(long millis, int nanos);
    public RelativeTime(RelativeTime time);

    public AbsoluteTime absolute(Clock clock, AbsoluteTime destination);

    public RelativeTime add(long millis, int nanos);
    public final RelativeTime add(RelativeTime time);

    public final RelativeTime subtract(RelativeTime time);

    ...
}

```

Program 9.5 The Real-Time Java Clock class.

```

public abstract class Clock {
    public Clock();

    public static Clock getRealtimeClock();

    public abstract RelativeTime getResolution();

    public AbsoluteTime getTime();
    public abstract void getTime(AbsoluteTime time);

    public abstract void setResolution(RelativeTime resolution);
}

```

9.2.3 Clocks in C/Real-Time POSIX

ANSI C has a standard library for interfacing to ‘calendar’ time. This defines a basic time type `time_t` and several routines for manipulating objects of type `time_t`; Program 9.6 defines some of these functions. C/Real-Time POSIX allows many clocks to be supported by an implementation. Each clock has its own identifier (of type `clockid_t`), and the IEEE standard requires that at least one clock be supported (`CLOCK_REALTIME`). A typical C interface to Real-Time POSIX clocks is illustrated by Program 9.7.

The value returned by a clock (via `clock_gettime`) is given by the structure `timespec`; `tv_sec` represents the number of seconds expired since 1 January 1970 and `tv_nsec` the additional number of nanoseconds (although it is shown as a long integer, the value taken by `tv_nsec` must always be less than 1 000 000 000 and non-negative – C provides no subtyping mechanism). C/Real-Time POSIX requires that the minimum resolution of `CLOCK_REALTIME` is 50 Hz (20 milliseconds); the function `clock_getres` allows the resolution of a clock to be determined.

9.2.4 Real-time and monotonic clocks

In the above discussions, two types of clocks have been identified: clocks that allow the absolute external (calendar) time to be obtained and monotonic clocks that allow the passage of time to be measured accurately. Table 9.2 summarizes the difference in the various clocks provided by Ada, C/Real-Time POSIX and Real-Time Java. Note the term ‘real-time’ clock has a slightly different meaning across the languages.

Program 9.6 An interface to ANSI C dates and time.

```
typedef ... time_t;

struct tm {
    int tm_sec;      /* seconds after the minute - [0, 61] */
                    /* 61 allows for 2 leap seconds */
    int tm_min;      /* minutes after the hour - [0, 59] */
    int tm_hour;      /* hour since midnight - [0, 23] */
    int tm_mday;      /* day of the month - [1, 31] */
    int tm_mon;       /* months since January - [0, 11] */
    int tm_year;      /* years since 1900 */
    int tm_wday;      /* days since Sunday - [0, 6] */
    int tm_yday;      /* days since January 1 - [0, 365] */
    int tm_isdst;     /* flag for alternate daylight savings time */
}; double difftime(time_t time1, time_t time2);
/* subtract two time values */
time_t mktime(struct tm *timeptr); /* compose a time value */
time_t time(time_t *timer);
/* returns the current time and if timer is not null */
/*it also places the time at that location */
```

Program 9.7 The C/Real-Time POSIX interface to clocks.

```
#define CLOCK_REALTIME ...;
#define CLOCK_PROCESS_CPUTIME_ID ...;
#define CLOCK_THREAD_CPUTIME_ID ...;

struct timespec {
    time_t tv_sec;    /* number of seconds */
    long tv_nsec;    /* number of nanoseconds */
};
typedef ... clockid_t;

int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);
int clock_getres(clockid_t clock_id, struct timespec *res);

int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);
int clock_getcpuclockid(pthread_t_t thread_id, clockid_t *clock_id);

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
/* Note, that a nanosleep return -1 if the sleep is interrupted */
/* by a signal. In this case, rmtp has the remaining sleep time */

int nano_nanosleep(clockid_t clock_id, int flags,
    const struct timespec *rqtp, struct timespec *rmtp);
/* if flag = TIMER_ABSTIME, then the sleep is absolute */
/* using the identified clock */
```

	Calendar	Monotonic
Ada	package Calendar	package Real_Time
C/Real-Time POSIX	time function CLOCK_REALTIME	CLOCK_MONOTONIC
Real-Time Java	currentTimeInMillis function	getRealtimeClock function

Table 9.2 Calendar and monotonic clocks.

9.3 Delaying a task

In addition to having access to a clock, tasks must also be able to delay their execution either for a relative period of time or until some absolute time in the future.

9.3.1 Relative delays

A relative delay enables a task to queue on a future event rather than busy-wait on calls to the clock. For example, the following Ada code shows how a task can loop waiting for 10 seconds to pass:

```
Start := Clock; -- from calendar
loop
    exit when (Clock - Start) > 10.0;
end loop;
```

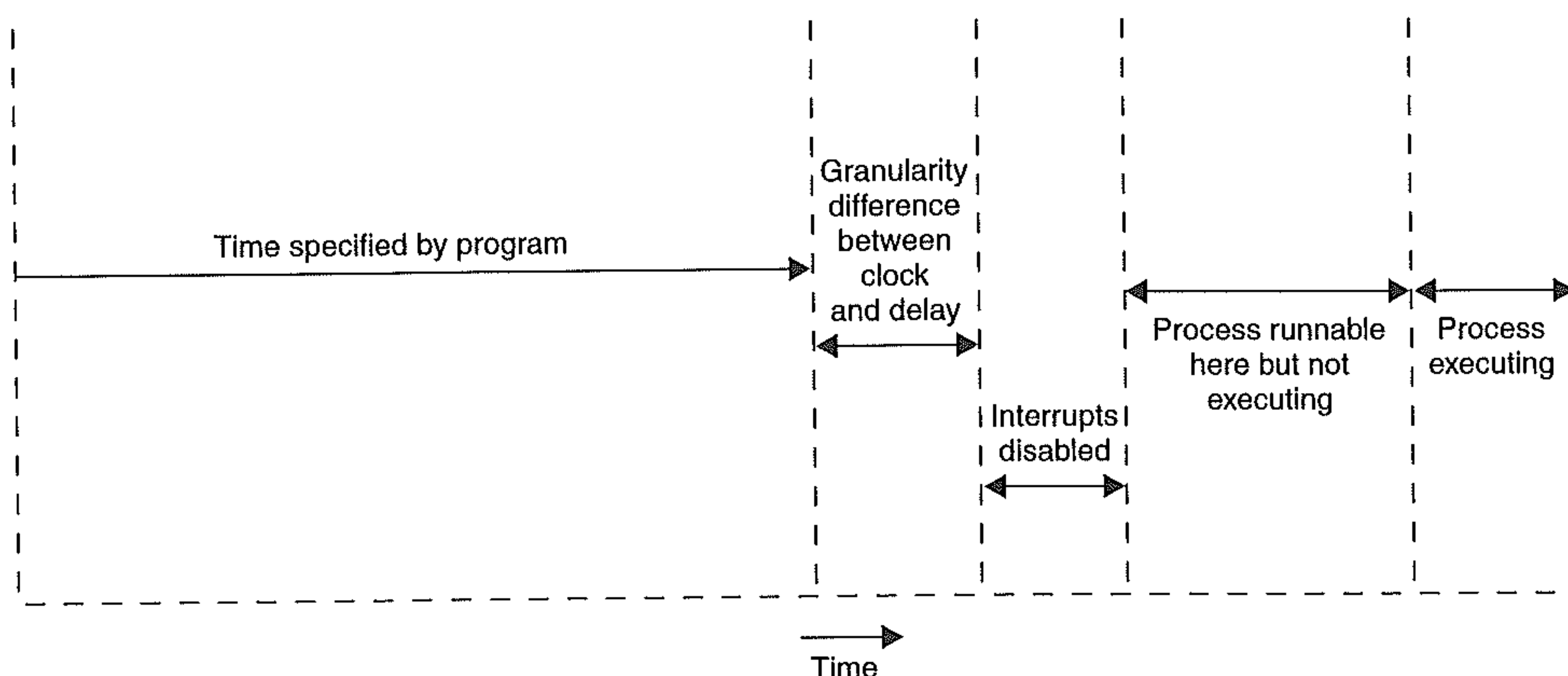


Figure 9.1 Delay times.

To eliminate the need for these busy-waits, most languages and operating systems provide some form of delay primitive. In Ada, this is a delay statement.

```
delay 10.0;
```

The value after **delay** (of type `Duration`) is relative (that is, the above statement means delay 10 seconds from the current time – a negative value is treated as zero).

In C/Real-Time POSIX, a delay can be obtained by use of the ‘sleep’ system call if a coarse granularity is required (that is, a delay of ‘seconds’), or ‘nanosleep’ if a finer granularity is required (see Program 9.7), the latter being measured in terms of the `CLOCK_REALTIME`. Java provides similar facilities to C/Real-Time POSIX. The `sleep` method in the `Thread` class allows a thread to delay itself at the milliseconds granularity. The `RealtimeThread` class allows a high-resolution sleep relative to a clock.

It is important to appreciate that a ‘delay’ or ‘sleep’ only guarantees that the task is made runnable after the period has expired.² The actual delay before the task begins executing is, of course, dependent on the other tasks which are competing for the processor. It should also be noted that the granularity of the delay and the granularity of the clock are not necessarily the same. For example, C/Real-Time POSIX and Real-Time Java allow a granularity down to the nanoseconds; however, few current systems will support this. Moreover, the internal clock may be implemented using an interrupt which could be inhibited for short periods. Figure 9.1 illustrates the factors affecting the delay.

9.3.2 Absolute delays

The use of **delay** in Ada (and `sleep` in C/Real-Time POSIX or Real-Time Java) supports a relative time period (for example, 10 seconds from now). If a delay to an absolute time is required, then either the programmer must calculate the period to

²In C/Real-Time POSIX, the process can actually be woken early if it receives a signal. In this case an error indication is returned, and the `rmpt` parameter returns the delay time remaining. The situation in Java is similar. If a thread is interrupted then it is awoken and the error object `InterruptedException` is thrown. In Ada, a delayed task is only woken early if it is within a `select-then-abort` statement.

delay or an additional primitive is required. For example, if an action should take place 10 seconds after the start of some other action, then the following Ada code could be used (with `Calendar`):

```
Start := Clock;
First_Action;
delay 10.0 - (Clock - Start);
Second_Action;
```

Unfortunately, this might not achieve the desired result. In order for this formulation to have the required behaviour, then

```
delay 10.0 - (Clock - Start);
```

would have to be an uninterruptible (atomic) action, which it is not. For example, if `First_Action` took two seconds to complete then

```
10.0 - (Clock - Start);
```

would equate to eight seconds. However, after having calculated this value, if the task involved is preempted by some other task it could be three seconds (say) before it next executes. At that time it will delay for eight seconds rather than five. To solve this problem, Ada introduces the `delay until` statement:

```
Start := Clock;
First_Action;
delay until Start + 10.0;
Second_Action;
```

As with `delay`, `delay until` is accurate only in its lower bound. The task involved will not be released before the current time has reached that specified in the statement, but it may be released later.

The time overrun associated with both relative and absolute delays is called the **local drift** and it cannot be eliminated. It is possible, however, to eliminate the **cumulative drift** that could arise if local drifts were allowed to superimpose. The following code, in Ada, shows how the computation `Action` is programmed to be executed, on average, every 7 seconds. This code will compensate for any local drift. For example, if two consecutive calls to `Action` were actually 7.4 seconds apart then the subsequent delay would be for only 6.6 seconds (approximately).

```
declare
    Next : Time;
    Interval : constant Duration := 7.0;
begin
    Next := Clock + Interval;
    loop
        Action;
        delay until Next;
        Next := Next + Interval;
    end loop;
end;
```

In Real-Time Java, the `sleep` method (defined in the `RealtimeThread` class) can be used for both relative and absolute delays.

Similarly, an absolute delay can also be constructed in C/Real-Time using the `clock_nanosleep` function (see Program 9.7).

9.4 Programming timeouts

Perhaps the simplest time constraint that an embedded system can have is the requirement to recognize, and act upon, the non-occurrence of some external event. For example, a temperature sensor may be required to log a new reading every second, the failure to give a reading within 10 seconds being defined as a fault. In general, a **timeout** is a restriction on the time a task is prepared to wait for a communication. Chapters 5 and 6 have discussed, in detail, inter-task communication facilities. Each one requires a timeout if it is to be used in a real-time environment.

As well as waiting for communication, timeouts are also required on the execution of actions. For example, a programmer might require that a section of code be executed within a certain time. If this does not happen at run-time, then some error recovery might be required.

9.4.1 Shared variable communication and timeouts

In Chapter 5, various communication and synchronization mechanisms based on shared variables were discussed. Both mutual exclusive access to a critical section and condition synchronization were seen as being important requirements. When a task attempts to gain access to a critical section, it is blocked if another task is already active in the section. This blocking, however, is bounded by the time taken to execute the code of the section and the number of other tasks that also wish to execute the critical section. For this reason, it is not usually deemed necessary to have a timeout associated with attempting entry. (However, C/Real-Time POSIX does provide such timeouts.) In Section 11.8, the issue of analysing this blocking time is considered in detail.

In contrast, the blocking associated with condition synchronization is not so easily bounded and depends on the application. For example, a producer task attempting to place data in a full buffer must wait for a consumer task to take data out. The consumer tasks might not be prepared to do this for a significant period of time. For this reason, it is important to allow tasks the option of timing out while waiting for condition synchronizations.

The condition synchronization facilities considered in Chapter 5 included:

- semaphores;
- conditional critical regions;
- condition variables in monitors, mutexes or synchronized methods;
- entries in protected objects.

Real-Time Euclid (Kligerman and Stoyenko, 1986) uses semaphores and extends the semantics of `wait` to include a time bound. The following statement illustrates how a process could suspend itself on the semaphore `CALL` with a timeout value of 10 seconds:

```
wait CALL noLongerThan 10 : 200
```


If the process is not signalled within 10 seconds, the exception 200 is raised (exceptions in Real-Time Euclid are numbered). Note, however, that there is no requirement for the process to be actually scheduled and executing within 10 seconds. The same is true for all timeout facilities. They merely indicate that the process should become runnable again.

C/Real-Time POSIX supports an explicit timeout for waiting on a semaphore or on a condition variable (see Programs 5.2 and 5.4). The following statement illustrates how a process could suspend itself on the semaphore `call` with an absolute timeout value of `timeout`:

```
if(sem_timedwait(&call, &timeout) < 0) {
    if ( errno == ETIMEDOUT) {
        /* timeout occurred */
    }
    else {
        /* some other error */
    }
} else {
    /* semaphore locked */
};
```

If the semaphore could not be locked by the `timeout`, `errno` is set to the value `ETIMEDOUT`.

With Ada's protected objects, avoidance synchronization is used and, therefore, it is at the point of the protected entry call that the timeout must be applied. As Ada treats a protected entry call the same as a task entry call, an identical mechanism for timeout is used. This is described below in terms of message passing.

With Java, the `wait` method can be used with a timeout either at millisecond granularity or at nanosecond granularity, the latter being provided by the `waitForObject` method in the `HighResolutionTime` class given in Program 9.3.

9.4.2 Message passing and timeouts

Chapter 6 has considered the various forms of message passing. All forms require synchronization. Even with asynchronous systems, a task may wish to wait for a message (or a sending task's message buffer might become full). With synchronous message passing, once a task has committed itself to a communication it must wait until such an event has occurred. For these reasons, timeouts are required. To illustrate the programming of timeouts consider a controller task (in Ada) that is called by some other driver task and given a new temperature reading:

```
task Controller is
    entry Call(T : Temperature);
end Controller;

task body Controller is
    -- declarations
begin
    loop
        accept Call(T : Temperature) do
            New_Temp := T;
        end Call;
```

```

    -- other actions
  end loop;
end Controller;

```

It is now required that the controller be modified so that the lack of the entry call is acted upon. This requirement can be provided using the constructs already discussed. A second task is used that delays itself for the timeout period and then calls the controller. However, the need for a timeout is so common that a more concise way of expressing it is desirable. This is usually provided in a real-time language as a special form of alternative in a selective wait. The above example would be coded as follows:

```

task Controller is
  entry Call(T : Temperature);
end Controller;

task body Controller is
  -- declarations
begin
  loop
    select
      accept Call(T : Temperature) do
        New_Temp := T;
      end Call;
    or
      delay 10.0;
      -- action for timeout
    end select;
    -- other actions
  end loop;
end Controller;

```

The delay alternative becomes ready when the time delay has expired. If this alternative is chosen (that is, a Call is not registered within 10 seconds) then the statements after the delay are executed.

The above example uses a relative delay, but absolute delays are also possible. Consider the following code that enables a task to accept registrations up to time Closing_Time:

```

task Ticket_Agent is
  entry Registration(...);
end Ticket_Agent;

task body Ticket_Agent is
  -- declarations
  Shop_Open : Boolean := True;
begin
  while Shop_Open loop
    select
      accept Registration(...) do
        -- log details
      end Registration;
    or
      delay until Closing_Time;
      Shop_Open := False;
    end select;
  end while;
end Ticket_Agent;

```



```

    -- task registrations
  end loop;
end Ticket_Agent;

```

Within the Ada model, it would not make sense to mix an else part, a terminate alternative and delay alternatives. These three structures are, therefore, mutually exclusive; a select statement can have, at most, only one of them. However, if it is a delay alternative, the select can have a number of delays but they must all be of the same kind (that is, all 'delay' statements or all 'delay until' statements). The one with the shortest duration or earliest absolute time is the operative one on each occasion.

A timeout facility is common in message-based concurrent programming languages. Ada actually goes further and allows a timeout on a message send. To illustrate this, consider the device driver that is feeding temperatures to the controller in the above Ada code:

```

loop
  -- get new temperature T
  Controller.Call(T);
end loop;

```

As new temperature readings are available continuously (and there is no point in giving the controller an out-of-date value), the device driver may wish to be suspended waiting for the controller for only half a second before withdrawing the call. This is achieved by using a special form of the select statement which has a single entry call and a single delay alternative:

```

loop
  -- get new temperature T
  select
    Controller.Call(T);
  or
    delay 0.5;
    null;
  end select;
end loop;

```

The **null** is not strictly needed but shows that again the delay can have arbitrary statements following, that are executed if the delay expires before the entry call is accepted. This is a special form of the select. It cannot have more than one entry call and it cannot mix entry calls and accept statements. The action it invokes is called a **timed entry call**. It must be emphasized that the time period specified in the call is a timeout value for the call being accepted; *it is not a timeout on the termination of the associated accept statement*.

When a task wishes only to make an entry call if the called task is immediately prepared to accept the call, then rather than make a timed entry call with time zero, a **conditional entry call** can be made:

```

select
  T.E  -- entry E in task T
else
  -- other actions
end select;

```

The ‘other actions’ are executed only if *T* is not prepared to accept *E* immediately. ‘Immediately’ means that either *T* is already suspended on **accept** *E* or on a select statement with such an open alternative (and it is chosen).

The above examples have used timeouts on inter-task communication; it is also possible, within Ada, to do timed (and conditional) entry call on protected objects:

```
select
  P.E ; -- E is an entry in Protected Object P
or
  delay 0.5;
end select;
```

9.4.3 Timeouts on actions

In Section 7.4, mechanisms were discussed that allowed tasks to have their control flows altered by asynchronous notifications. A timeout can be considered such a notification, and therefore if asynchronous notifications are supported, timeouts can also be used. Both a resumption (asynchronous events) and a termination (asynchronous transfer of control) model can be augmented by timeouts. Although for timeouts on actions, it is the termination model that is needed.

For example, in Section 7.6.1, the Ada asynchronous transfer of control (ATC) facility was introduced. With this, an action can be aborted if a ‘triggering event’ occurs before the action has completed. One of the allowed triggering events is the passage of time. To illustrate this, consider a task that contains an action that must be completed within 100 milliseconds. The following code supports this requirement directly:

```
select
  delay 0.1;
then abort
  -- action
end select;
```

If the action takes too long, the triggering event will be taken and the action will be aborted. This is clearly an effective way of catching code that is stuck in a non-terminating loop or has some other program error.

Timeouts are thus usually associated with error conditions; if a communication has not occurred within *X* milliseconds then something unfortunate has happened and corrective action must be taken. This is, however, not their only use. Consider a task that has a compulsory component and an optional part. The compulsory computations produce (quickly) an adequate result that is assigned to, say in Ada, a protected object. The task must complete by a fixed time; but if there is time available after the compulsory component has been completed, an optional algorithm can be used to incrementally improve the output value. To program this again requires a timeout on an action.

```
declare
  Precise_Result : Boolean;
begin
  Completion_Time := ...
  -- compulsory part
  Results.Write(...); -- call to procedure in
                      -- external protected object
```

Program 9.8 The Real-Time Java Timed class.

```

public class Timed extends AsynchronouslyInterruptedException
    implements java.io.Serializable
{
    public Timed(HighResolutionTime time)
        throws IllegalArgumentException;

    public boolean doInterruptible(Interruptible logic);

    public void resetTime(HighResolutionTime time);
}

```

```

select
    delay until Completion_Time;
    Precise_Result := False;
then abort
    while Can_Be_Improved loop
        -- improve result
        Results.Write(...);
    end loop;
    Precise_Result := True;
end select;
end;

```

Note that if the timeout expires during the write to the protected object it will complete its write correctly, as a call to a protected object is an abort-deferred action (that is, the effect of the abort is postponed until the task leaves the protected object).

With Real-Time Java, timeouts on actions are provided by a subclass of `AsynchronouslyInterruptedException` called `Timed`. This class is defined in Program 9.8 (see Section 6.8.4 for information concerning `java.io.Serializable`).

The above Ada example would be fully written in Real-Time Java as follows:

```

public class PreciseResult {
    public resultType value; // the result
    public boolean preciseResult; // indicates if it is imprecise
}

public class ImpreciseComputation {
    private HighResolutionTime CompletionTime;
    private PreciseResult result = new PreciseResult();

    public ImpreciseComputation(HighResolutionTime T)
    {
        CompletionTime = T; //can be absolute or relative
    }

    private resultType compulsoryPart()
    {
        // function which computes the compulsory part
    }
}

```

```

public PreciseResult Service() // public service
{
    Interruptible I = new Interruptible()
    {
        public void run(AsynchronouslyInterruptedException exception)
            throws AsynchronouslyInterruptedException
        {
            // this is the optional function which improves on the
            // compulsory part
            boolean canBeImproved = true;

            while(canBeImproved)
            {
                // improve result
                synchronized(this) {
                    // write result --
                    // the synchronized statement ensures
                    // atomicity of the write operation
                }
            }
            result.preciseResult = true;
        }

        public void interruptAction(
            AsynchronouslyInterruptedException exception)
        {
            result.preciseResult = false;
        }
    };

    Timed t = new Timed(CompletionTime);

    result.value = compulsoryPart(); // compute the compulsory part
    if(t.doInterruptible(I)) {
        // execute the optional part with the timer
        return result;
    } else { ... };
}

```

Timeouts are an important feature of real-time systems; they are, however, far from being the only time constraints of significance. The rest of this chapter, and the following one, deals with the more general topic of time deadlines and how to ensure that they are met.

9.5 Specifying timing requirements

For many important real-time systems, it is not sufficient for the software to be logically correct; the programs must also satisfy timing constraints determined by the underlying physical system. These constraints can go far beyond simple timeouts. Unfortunately, existing practices in the engineering of large real-time systems are, in general, still rather

ad hoc. Often, a logically correct system is specified, designed and constructed (perhaps as a prototype) and then tested to see if it meets its timing requirements. If it does not, then various fine tunings and rewrites ensue. The result is a system that may be difficult to understand and expensive to maintain and upgrade. A more systematic treatment of time is required.

Work on a more rigorous approach to this aspect of real-time systems has followed two largely distinct paths. One direction of development has concerned the use of formally defined language semantics and timing requirements, together with notations and logics that enable temporal properties to be represented and analysed. The other path has focused on the performance of real-time systems in terms of the feasibility of scheduling the required workload on the available resources (processors and so on).

In this book, attention is focused mainly on the latter work. The reasons for this are three-fold. Firstly, the formal techniques are not yet mature enough to reason about large complex real-time systems. Secondly, there is little reported experience of the use of these techniques in actual real-time systems. Finally, to include a full discussion of such methods would involve a substantial amount of material that is outside the scope of this book. This is not meant to imply that the area is irrelevant to real-time systems. The understanding of, for example, formal techniques based on Communicating Sequential Processes (CSP), temporal logics, real-time logics, model checking and specification techniques that incorporate notions of time is becoming increasingly important.

The success of model checking in verifying functional properties has recently been extended into the real-time domain. A system is modelled as a Timed Automaton (that is, a finite state machine with clocks) and then model checking is used to 'prove' that undesirable states cannot be reached or that desirable states will be entered before some internal clock has reached a critical time (deadline). The latter property is known as *bounded liveness*. Although the explorations undertaken by model checking are subject to state explosion, a number of tools are now available that can tackle problems of a reasonable size. The technique is likely to become standard industrial practice over the coming years.

The verification of a real-time system can usefully be interpreted as requiring a two-stage process.

- (1) Verifying requirements/designs – given an infinitely fast reliable computer, are the temporal requirements coherent and consistent; that is, have they the potential to be satisfied?
- (2) Verifying the implementation – with a finite set of (possible unreliable) hardware resources, can the temporal requirements be satisfied?

As indicated above, (1) may require formal reasoning (and/or model checking) to verify that necessary temporal (and causal) orderings are satisfied. For example, if event A must be completed before event B, but is dependent on some event C that occurs after B, then no matter how fast the processor it will never be possible to satisfy these requirements. Early recognition of this difficulty is therefore very useful. The second issue (implementation verification) is the topic of Chapter 11. The remainder of this chapter will concentrate on how temporal requirements can actually be represented in languages.

9.6 Temporal scopes

To facilitate the specification of the various timing constraints found in real-time applications, it is useful to introduce the notion of **temporal scopes**. Such scopes identify a collection of statements with an associated timing constraint. The possible attributes of a temporal scope (TS) are illustrated in Figure 9.2, and include:

- (1) **deadline** – the time by which the execution of a TS must be finished;
- (2) **minimum delay** – the minimum amount of time that must elapse before the start of execution of a TS;
- (3) **maximum execution time** – of a TS;
- (4) **maximum elapsed time** – of a TS.

Temporal scopes with combinations of these attributes are also possible, and for some timing constraints a combination of sequentially executed temporal scopes is necessary. For example consider a simple control action that reads a sensor, computes a new setting and outputs this setting via an actuator. To get fine control over when the sensor is read, an initial temporal scope with a tight deadline is needed. The output is produced in a second temporal scope which has a minimum delay equal to the first

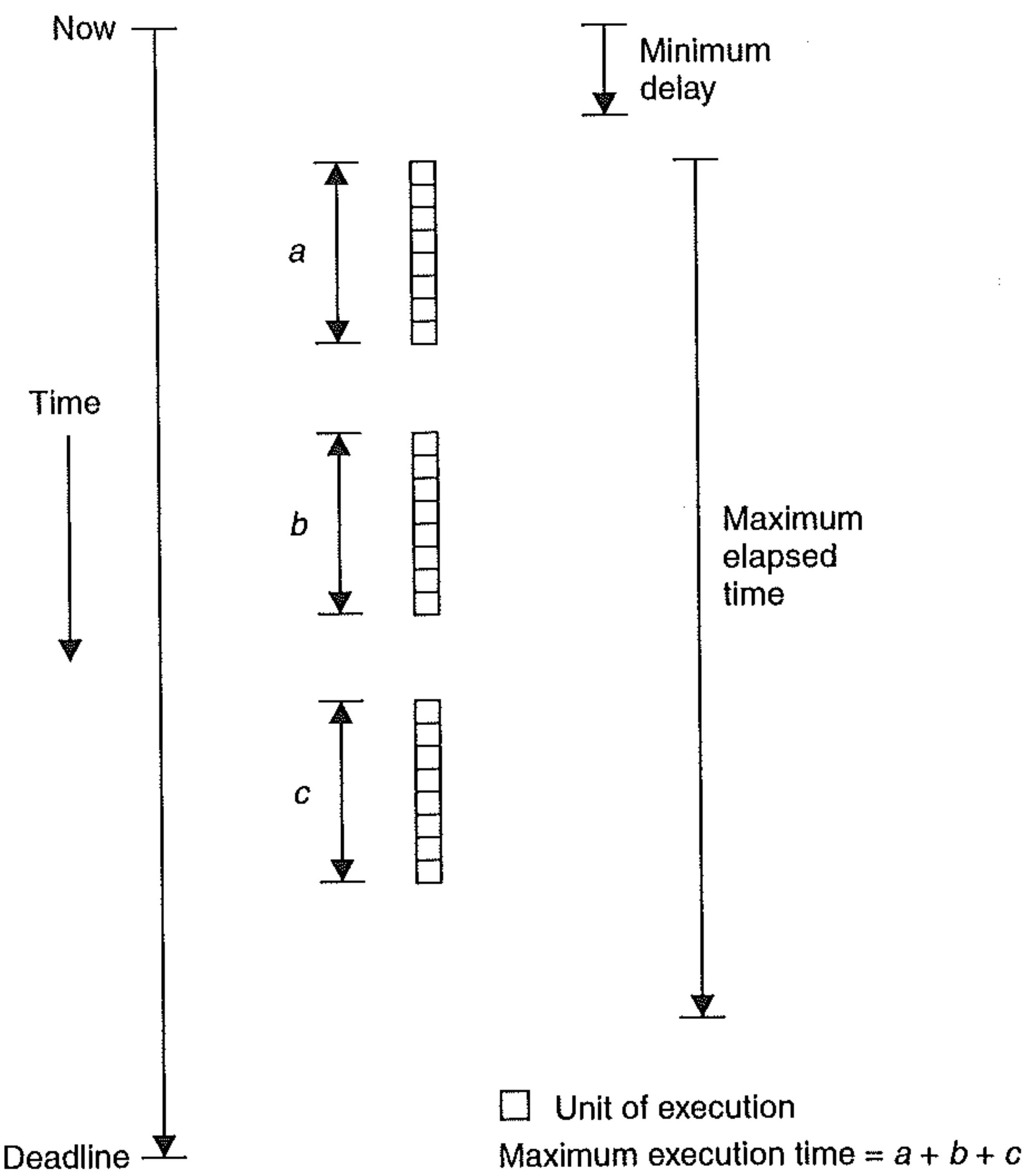


Figure 9.2 Temporal scopes.

scope's deadline but a later deadline itself. The variation of when a sensor is read is called **input jitter**. If there is a need to also control **output jitter** then a third temporal scope could be added which has a long 'minimum delay' and a short time interval before its deadline.

Temporal scopes can themselves be described as being either **periodic** or **aperiodic**. Typically, periodic temporal scopes sample data or execute a control loop and have explicit deadlines that must be met. Aperiodic, or **sporadic**, temporal scopes usually arise from asynchronous events outside the embedded computer. These scopes have specified response times associated with them.

In general, aperiodic temporal scopes are viewed as being activated randomly, following, for example, a Poisson distribution. Such a distribution allows for bursts of arrivals of external events, but does not preclude any possible concentration of aperiodic activity. It is therefore not possible to do worst-case analysis (there is a non-zero probability of any number of aperiodic events occurring within a given time). To allow worst-case calculations to be made, a minimum period between any two aperiodic events (from the same source) is often defined. If this is the case, the task involved is said to be **sporadic**. In this book, the term 'aperiodic' is used for the general case and 'sporadic' is reserved for situations where a minimum delay between subsequent executions is needed.

In many real-time languages, temporal scopes are, in effect, associated with the tasks that embody them. Tasks can be described as either periodic, aperiodic or sporadic depending on the properties of their internal temporal scope. Most of the timing attributes given in the above list can thus be satisfied by:

- (1) running periodic tasks at the correct rate;
- (2) completing all tasks by their deadline.

The problem of satisfying timing constraints thus becomes one of scheduling tasks to meet deadlines, or **deadline scheduling**.

Although all computer systems strive to be efficient, and many are described as real-time, further classification is needed to deal adequately with the different levels of importance that time has within applications. As noted in Chapter 1, a system is said to be **hard** real-time if it has deadlines that cannot be missed or else the system fails. By comparison, a system is **soft** if the application is tolerant of missed deadlines. A system is merely **interactive** if it does not have specified deadlines but strives for 'adequate response times'. To give a little more precision, a task with a soft deadline may still deliver its service late – the system will still draw some value from the tardy service. A non-hard task that has a fixed rigid deadline (that is, a tardy service is useless/valueless) is said to be **firm**.

The distinction between hard, firm and soft real-time becomes somewhat blurred in fault-tolerant systems. Nevertheless, it is usually appropriate to use the term 'hard' if a specific error recovery (or fail safe) routine is triggered by a missed deadline, and firm/soft if the nature of the application is tolerant of the occasional missed deadline or deadlines that are not missed by much. Finally, note that many hard real-time systems will have some deadlines which are soft or firm.

9.6.1 Specifying tasks and temporal scopes

In real-time systems, it is necessary to deal explicitly with timing requirements, four types of which were given earlier. A general scheme for a periodic task is thus as follows:

```
task Task_T;
...
begin
  loop
    IDLE
    start of temporal scope
    ...
    end of temporal scope
  end;
end;
```

The time constraints take the form of a minimum time for IDLE and the requirement that the end of the temporal scope be by some deadline. This deadline can itself be expressed in terms of either:

- absolute time;
- execution time since the start of the temporal scope; or
- elapsed time since the start of the temporal scope.

As noted earlier a task that is sampling data may be composed of a number of temporal scopes:

```
loop
  start of 1st temporal scope
  ...
  end of 1st temporal scope
  start of 2nd temporal scope
  ...
  end of 2nd temporal scope
  IDLE
  start of 3rd temporal scope
  ...
  end of 3rd temporal scope
end;
```

The input activities of this task take place in the 1st temporal scope and hence the deadline at the end of this scope regulates the maximum input jitter for the task. The input data may already be available in buffers for the task, or the task may need to read input registers in the sensor's device interface. The 2nd temporal scope incorporates whatever computations are needed to calculate the task's output values (this may include interacting with other tasks). The 3rd scope is concerned with the output action. Here the IDLE interval is important; it is measured from the beginning of the loop and constrains the time before the output can be produced by the task. The deadline on this final temporal scope places an upper bound on the time of the output phase.

Although it would be possible to incorporate this sequence into a single task, scheduling analysis (see Chapter 11) places restrictions on the structure of a task.

Specifically, a task must only have at most one IDLE statement (at the beginning of the execution sequence) and one deadline (at the end). So, for illustration, assume the control task has a period of 100 ms, a constraint on input jitter of 5 ms, a constraint on output jitter of 10 ms and a deadline of 80 ms. The three necessary tasks would take the form:

```
task periodic_PartA;
...
begin
  loop every 100ms
    start of temporal scope
      input operations
      write data to a shared object
    end of temporal scope - deadline 5ms
  end;
end;

task periodic_PartB;
...
begin
  loop every 100ms
    IDLE 5ms
    start of temporal scope
      read form shared object
      computations
      write data to a shared object
    end of temporal scope - deadline 65ms
  end;
end;

task periodic_PartC;
...
begin
  loop every 100ms
    IDLE 70ms
    start of temporal scope
      read form shared object
      output operations
    end of temporal scope - deadline 10ms
  end;
end;
```

Deadlines are also desirable with aperiodic tasks; here the temporal scope is triggered by an external event that will normally take the form of an interrupt:

```
task aperiodic_P;
...
begin
  loop
    wait for interrupt
    start of temporal scope
      ...
    end of temporal scope
  end;
end;
```

Clearly, a periodic task has a defined periodicity (that is, how often the task loop is executed); this measure may also be applied to an aperiodic task in which case it means the maximum rate at which this task will cycle (that is, the fastest rate for interrupt arrivals). As stated earlier, such aperiodic tasks are known as sporadic.

Summary

The management of time presents a number of difficulties that set embedded systems apart from other computing applications. Current real-time languages are often inadequate in their provisions for this vital area.

The introduction of the notion of time into real-time programming languages has been described in terms of four requirements:

- access to a clock;
- delaying;
- timeouts;
- deadline specification and scheduling.

The sophistication of the means provided to measure the passage of time varies greatly between languages. Ada and Real-Time Java provide two abstract data types for time and a collection of time-related operators. C/Real-Time POSIX provides a comprehensive set of facilities for clocks and timers including periodic timers.

If a task wishes to pause for a period of time, a delay (or sleep) primitive is needed to prevent the task having to busy-wait. Such a primitive always guarantees to suspend the task for at least the designated time, but it cannot force the scheduler to run the task immediately the delay has expired. It is not possible, therefore, to avoid local drift, although it is possible to limit the cumulative drift that could arise from repeated execution of delays.

For many real-time systems, it is not sufficient for the software to be logically correct; the programs must also satisfy timing constraints. Unfortunately, existing practices in the engineering of large real-time systems are, in general, still rather *ad hoc*. To facilitate the specification of timing constraints and requirements, it is useful to introduce the notion of a 'temporal scope'. Possible attributes of temporal scopes include:

- deadline for completion of execution;
- minimum delay before start of execution;
- maximum execution time;
- maximum elapse time.

Temporal scopes can be combined and embedded in tasks that are required to execute in accordance with their timing constraints. Input and output jitter can be controlled and deadlines specified for the actions necessary to produce inputs from outputs.

Further reading

- Buttazzo, G. C. (1997) *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. New York: Kluwer Academic.
- Coulouris, G., Dollimore, J. and Kindberg, T. (2005) *Distributed Systems, Concepts and Design* (4th edn). Harlow: Addison-Wesley.
- Joseph, M. (ed.) (1996) *Real-Time Systems: Specification, Verification and Analysis*. Englewood Cliffs, NJ: Prentice Hall.
- Koptez, H. (1997) *Real-Time Systems*. New York: Kluwer Academic.
- Turski, W. M. (1988) Time considered irrelevant for real-time systems. *BIT*, **28**(3), 473–488.

Exercises

- 9.1 Explain how a system can be transformed so that all timing failures manifest themselves as value failures. Can the converse be achieved?
- 9.2 Should Ada's timed entry call specify a timeout on the *completion* of the rendezvous, rather than the start of the rendezvous? Give an example of when such an approach might be useful. How might the same effect be obtained?

Chapter 10

Programming real-time abstractions

10.1	Real-time tasks	10.5	Controlling input and output jitter
10.2	Programming periodic activities	10.6	Other approaches for supporting temporal scopes
10.3	Programming aperiodic and sporadic activities		Summary
10.4	The role of real-time events and their handlers		Further reading
			Exercises

One of the issues raised in the previous chapter was the need to specify timing requirements arising from within the application. The notion of a *temporal scope* was introduced. A temporal scope has several attributes that when combined within a task allow important real-time characteristics to be presented. This chapter illustrates how combinations of these attributes can be used to:

- program periodic tasks;
- program sporadic and aperiodic tasks; and
- control input and output jitter.

Tasks that have any of the above attributes are termed **real-time tasks**.

Languages can support the programming of real-time tasks at various levels of abstraction. For example, Ada and C/Real-Time POSIX provide relatively low-level abstractions that the programmer can combine to implement various real-time models. In contrast, Real-Time Java provides a combination of low- and high-level abstraction. Research-oriented languages tend to focus on particular aspects and explore language support in those areas.

This chapter first considers the relationship between a real-time task and the typical task representation techniques that were considered in Chapter 4. It then illustrates how periodic activities can be programmed in C/Real-Time POSIX, Ada and Real-Time Java. This is followed by a discussion on how event-triggered sporadic and aperiodic activities can be implemented in the same languages. After this, the chapter turns its attention to the control of input and output jitter. Following this, brief consideration is given to the other languages to illustrate the alternative support that could be given.

10.1 Real-time tasks

In theory any concurrent activity (be it a result of explicit task creation or the execution of a fork or a cobegin statement) can be considered real-time if real-time attributes are associated with it, and it is scheduled for execution by a real-time scheduler. However, in practice, real-time tasks often have constraints placed on the programming style to ensure that their execution is predictable – if not deterministic. For example, a hard real-time task might be prohibited from executing an unbounded loop statement. Languages like Ada and C/Real-Time POSIX do not syntactically distinguish between a concurrent task/thread and a real-time task/thread. Instead, programming guidelines are applied and tools in the development environment are used to check conformance.

Real-Time Java takes a slightly different approach because it is an extension of Java and consequently must allow the execution of Java programs with standard Java semantics. For this reason, it distinguished between Java threads and those threads that have real-time semantics. The latter are called real-time threads, and are represented by a standard class in the Real-Time Java environment – shown in Programs 10.1 and 10.2. Real-time threads in Java have the following attributes associated with them.

- **Release parameters** – giving, amongst other things, the real-time thread's deadline; if the object is released periodically or sporadically, then subclasses allow an interval to be given. The base class for release parameters is given in Program 10.3. As well as having a deadline, all releases have an associated 'cost'. This is typically the worst-case execution time of the real-time thread (see Section 11.13).

Program 10.1 An extract of the RealtimeThread class.

```
package javax.realtime;
public class RealtimeThread extends Thread
    implements Schedulable {
    // constructors
    public RealtimeThread();
    public RealtimeThread(SchedulingParameters scheduling);
    public RealtimeThread(SchedulingParameters scheduling,
        ReleaseParameters release);
    public RealtimeThread(SchedulingParameters scheduling,
        ReleaseParameters release, MemoryParameters memory,
        MemoryArea area, ProcessingGroupParameters group,
        Runnable logic);

    // methods
    public void release(); // used for aperiodic execution
    public void start(); // override Thread.start()
    public boolean waitForNextPeriod();
        // used for periodic execution
    public boolean waitForNextRelease();
        // used for aperiodic execution
    ...
}
```

Program 10.2 An extract of the NoHeapRealtimeThread class.

```

package javax.realtime;
public class NoHeapRealtimeThread extends RealtimeThread {
    // constructors

    public NoHeapRealtimeThread(
        SchedulingParameters scheduling, MemoryArea area);
    public NoHeapRealtimeThread(
        SchedulingParameters scheduling,
        ReleaseParameters release, MemoryArea area);
    ...
}

```

Program 10.3 The ReleaseParameters class.

```

package javax.realtime;
public class ReleaseParameters implements Cloneable{
    // constructors
    protected ReleaseParameters();
    protected ReleaseParameters(RelativeTime cost,
        RelativeTime deadline,
        AsyncEventHandler overrunHandler,
        AsyncEventHandler missHandler);

    protected ReleaseParameters(RelativeTime cost,
        RelativeTime deadline,
        RelativeTime blockingTerm,
        AsyncEventHandler overrunHandler,
        AsyncEventHandler missHandler);
    // methods
    public Object clone();
    public RelativeTime getCost();
    public AsyncEventHandler getCostOverrunHandler();
    public RelativeTime getDeadline();
    public AsyncEventHandler getDeadlineMissHandler()
    public void setCost(RelativeTime cost);

    public void setCostOverrunHandler(
        AsyncEventHandler handler);
    public void setDeadline(RelativeTime deadline);

    public void setDeadlineMissHandler(
        AsyncEventHandler handler);
    public boolean setIfFeasible(RelativeTime cost,
        RelativeTime deadline);

    public RelativeTime getBlockingTerm();
    public void setBlockingTerm(RelativeTime blockingTerm);
    public boolean setIfFeasible(RelativeTime cost,
        RelativeTime deadline, RelativeTime blockingTerm);
}

```

- **Scheduling parameters** – defining the attributes that will be used by the real-time scheduler (see Section 12.7).
- **Memory parameters** – giving the maximum amount of memory used by the object in its default memory area, the maximum amount of memory used in immortal memory, and a maximum allocation rate of heap memory (see Section 14.7.1).
- **Processing group parameters** – this allows several schedulable objects to be treated as a group and to have an associated period, cost and deadline (see Section 13.6.2).

Further information about these parameter classes will be given in due course. For the time being it is adequate to know that there are default values that can be overwritten via the various constructors shown in Program 10.1.

In spite of this explicit identification of a real-time thread, Real-Time Java provides little support to enforce hard real-time programming guidelines. The exception is that it defines a subclass of the `RealtimeThread` class called `NoHeapRealtimeThread` – see Program 10.2. Threads created via this class are not allowed to access the Java heap and therefore cannot be subject to garbage collection delay.¹

10.2 Programming periodic activities

10.2.1 Ada and C/Real-Time POSIX

In keeping with many real-time languages, Ada and C/Real-Time POSIX do not support the explicit specification of periodic or sporadic tasks with deadlines; rather low-level mechanisms are provided that can be used for a variety of purposes – e.g. a delay primitive, timers and so on. These can be used within a looping task/thread to provide the same functionality as a periodic activity.

For example, in Ada, a periodic task might take the following form:

```
task body Periodic_T is
  Next_Release : Time;
  Release_Interval : constant Time_Span := Milliseconds(...);
begin
  -- read clock and calculate the next
  -- release time (Next_Release)
  loop
    -- sample data (for example) or
    -- calculate and send a control signal
    delay until Next_Release;
    Next_Release := Next_Release + Release_Interval;
  end loop;
end Periodic_T;
```

¹Hard real-time systems, particularly those that are safety-critical, are very conservative in their use of dynamic memory, and prefer not to rely on garbage collection due to the potential for unpredictable delays – see Section 14.7.1.

This is comparable to the C/Real-Time POSIX representation:

```
#include <signal.h>
#include <time.h>
#include <pthread.h>
#include <misc/timespec_operations.h>
void periodic_thread() /* destined to be the thread */
{
    struct timespec next_release, remaining_time;
    struct timespec thread_period; /* actual period */

    /* read clock and calculate the next
       release time (next_release) */

    while(1) {
        /* sample data (for example) or
           calculate and send a control signal */

        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                        &next_release, &remaining_time);
        add_timespec(&next_release, &next_release, &thread_period);
    }
}

int init(){
    pthread_attr_t attributes;      /* thread attributes */
    pthread_t PT;                  /* thread pointer */

    pthread_attr_init(&attributes); /* default attributes */
    pthread_create(&PT, &attributes,
                  (void *)periodic_thread, (void *)0);
}
```

The `clock_nanosleep` function is equivalent to Ada's 'delay until' statement. In this case, it uses the real-time clock and an absolute time. Note that the sleep can be interrupted by a signal, in which case `remaining_time` indicates how long is remaining. In this example, it is assumed that no interruption occurs.

As time values in C/Real-Time POSIX are structures, the `add_timespec` function is used to add two time values together and return the result in a third value.

10.2.2 Real-Time Java

In Real-Time Java, periodic activities are represented by real-time threads with associated periodic release parameters. The `PeriodicParameters` class is given in Program 10.4. In addition to the release parameter attributes, each periodic activity has a start time (which may be relative or absolute) and a period.

The method `start` is called for the initial release of the thread; once the thread has executed, it calls `waitForNextPeriod` (wFNP) to indicate to the scheduler that

Program 10.4 An extract of the `PeriodicParameters` class.

```

package javax.realtime;
public class PeriodicParameters extends ReleaseParameters {
    // Constructors: throw IllegalArgumentException if
    // period is null.
    public PeriodicParameters(RelativeTime period);
    public PeriodicParameters(HighResolutionTime start,
        RelativeTime period);
    public PeriodicParameters(
        HighResolutionTime start, RelativeTime period,
        RelativeTime cost, RelativeTime deadline,
        AsyncEventHandler overrunHandler,
        AsyncEventHandler missHandler);

    public PeriodicParameters(
        HighResolutionTime start, RelativeTime period,
        RelativeTime cost, RelativeTime deadline,
        RelativeTime blockingTerm,
        AsyncEventHandler overrunHandler,
        AsyncEventHandler missHandler);

    // methods
    public RelativeTime getPeriod();
    public HighResolutionTime getStart();
    public void setPeriod(RelativeTime period);
    public void setStart(HighResolutionTime start);
    ...
}

```

it should be made executable again when its next period is due. The following illustrates the approach:

```

public class Periodic extends RealtimeThread {
    public Periodic(PeriodicParameters P)
    { ... }

    public void run() {
        boolean deadlineMet = true;
        while(deadlineMet) {
            // code to be run each period
            ...
            deadlineMet = waitForNextPeriod();
        }
    }
}

```

The actual semantics of wFNP are quite complex and depend on whether the real-time thread has missed its deadline and whether the programmer has requested that an asynchronous event handler be released when this occurs (see Section 13.2.2). Here, it

is assumed that no handler is to be released, in which case the semantics of wFNP are as follows.

- If the deadline was met – wFNP returns true at the next release time.
- If the deadline was missed – wFNP return false immediately it is called. Effectively this allows the programmer to respond to the missed deadline and call wFNP again. The second call to wFNP returns true at the next release time.

The following program fragment illustrates how to create a periodic thread with a 10 millisecond period, a 5 millisecond deadline, whose first execution is delayed until an absolute time A. The thread should consume no more than 1 millisecond of processor time.

```
{
  AbsoluteTime A = new AbsoluteTime(...);
  PeriodicParameters P = new PeriodicParameters(
    A, // start time
    new RelativeTime(10,0), // period
    new RelativeTime(1,0), // cost
    new RelativeTime(5,0), // deadline
    null, null ); // no deadline miss/cost overrun handlers

  Periodic ourThread = new Periodic(P); //create thread
  ourThread.start(); // release it
}
```

Note that as with Ada and C/Real-time POSIX, it is necessary to code the loop. However, unlike these languages, the program does not explicitly have to calculate the required delay.

10.3 Programming aperiodic and sporadic activities

As indicated in Section 9.6, the main difference between aperiodic and sporadic activities is that the latter have a defined minimum inter-arrival time (MIT), whereas the former do not. From a programming point of view, the main issue is whether the language/operating system supports the detection of MIT violations. This will be considered in Section 13.7.1. Here the focus is on the structure needed to support general aperiodic tasks.

10.3.1 Aperiodic tasks in Ada

In common with Ada's approach to supporting periodic task, the language provides no direct abstractions to support aperiodic or sporadic tasks. Instead, the lower-level mechanisms can be used to program the precise model required. Consider, for example, an aperiodic Ada task that is triggered by an interrupt. Typically, it is necessary to use a protected object to handle the interrupt and release the task for execution:

```
protected Aperiodic_Controller is
  procedure Interrupt; -- mapped onto interrupt
```

```

    entry Wait_For_Next_Interrupt;
private
    Call_Outstanding : Boolean := False;
end Aperiodic_Controller;

protected body Aperiodic_Controller is
    procedure Interrupt is
    begin
        Call_Outstanding := True;
    end Interrupt;

    entry Wait_For_Next_Interrupt when Call_Outstanding is
    begin
        Call_Outstanding := False;
    end Wait_For_Next_Interrupt;
end Aperiodic_Controller;

task body Aperiodic_T is
begin
    loop
        Aperiodic_Controller.Wait_For_Next_Interrupt;
        -- action
    end loop;
end Aperiodic_T;

```

The details of the Ada model of interrupt handling will be considered in Section 14.3. Here it is assumed that the occurrence of an interrupt results in a call to the procedure `Interrupt`. This simply sets the boolean variable to `True`, thereby opening the guard and allowing the task to be released. As with periodic tasks, a loop must be used to get repeated releases.

10.3.2 Aperiodic threads and C/Real-Time POSIX

In C/Real-Time POSIX, if the aperiodic thread is to be released by another thread, then a similar approach to the one given above for Ada can be used, that is a monitor can be created with two methods, `wait_for_release` and `release`. The aperiodic thread is encapsulated in a loop. It calls `wait_for_release` where it is held on a condition variable until notified by a call to `release`.

Interrupts cannot be handled directly by the application. Instead the OS generates a signal. The aperiodic thread has a similar structure, but instead of calling a monitor method it suspends itself using the `sig_suspend` method described in Section 7.5.1. The thread is released by the occurrence of the signal.

10.3.3 Aperiodic threads and Real-Time Java

Unlike Ada and C/Real-Time POSIX, Real-Time Java directly supports the abstractions of aperiodic and sporadic activities. Aperiodic activity can occur at any time. Hence a schedulable object with `AperiodicParameters` can give no added information to its release characteristics over that supplied by the `ReleaseParameters`. However, as it is likely that a release will occur before the previous release has completed, an

Program 10.5 An abridged version of the `AperiodicReleaseParameters` class.

```

package javax.realtime;
public class AperiodicParameters extends ReleaseParameters {
    // fields
    public static final String arrivalTimeQueueOverflowExcept;
        //throw an exception in the releasing real-time thread
    public static final String arrivalTimeQueueOverflowIgnore;
        // ignore the release call
    public static final String arrivalTimeQueueOverflowReplace;
        // replace the last release call with the current one
    public static final String arrivalTimeQueueOverflowSave;
        // extend the queue and save the request
    // constructors
    public AperiodicParameters();
    public AperiodicParameters(
        RelativeTime cost, RelativeTime deadline,
        AsyncEventHandler overrunHandler,
        AsyncEventHandler missHandler);

    // methods
    public String getArrivalTimeQueueOverflowBehavior();
    public void setArrivalTimeQueueOverflowBehavior(
        String behavior);
    public int getInitialArrivalTimeQueueLength();
    public void setInitialArrivalTimeQueueLength(
        int initial);
    ...
}

```

implementation will maintain an internal queue of outstanding invocation requests for aperiodic schedulable objects. Consequently, Real-Time Java provides facilities to define the size of the queue and what happens if the queue overflows. The class definition is given in Program 10.5.

Given this support, the structure of an aperiodic real-time thread in Real-Time Java is similar in structure to that of a periodic one, and is shown below:

```

public class Aperiodic extends RealtimeThread {
    public Aperiodic(PriorityParameters PP, AperiodicParameters P)
    { ... }

    public void run() {
        boolean deadlineMet = true;

        while(deadlineMet) {
            // code to be run each period
            ...
            deadlineMet = waitForNextRelease();
        }
    }
}

```

Instead of calling the `waitForNextPeriod` method as in Section 10.2.2, the real-time thread calls the `waitForNextRelease` method. Its semantics are similar, except that the release event is when another real-time thread calls the `Release` method in the associated class shown in Program 10.1.

It should be noted that Real-Time Java interrupt handlers are second-level handlers and release an asynchronous event handler (see Section 14.4.2).

10.4 The role of real-time events and their handlers

It is useful in real-time systems to distinguish between two forms of computation that occur at run-time: tasks and event handlers. A real-time task (or thread) is a long-lived entity with state and periods of activity and inactivity. While active, it competes with other tasks for the available resources – the rules of this competition are captured in a scheduling or dispatching policy (for example, fixed priority or EDF). A real-time task can be released by the passage of time or by an event, examples of which have been given in the previous sections.

Event handlers can similarly be released by the passage of time or be event triggered. However, in comparison with real-time tasks, an event handler is usually a short-lived, stateless, one-shot or periodic computation. Its execution is, at least conceptually, immediate; and having completed it has no lasting direct effect other than by means of changes it has made to the permanent state of the system. For example, if a central heating system must come on at 7.00 am then the control system needs a clock and a way of postponing execution until that clock says 7.00 am. Using tasks, the only way to deliver this coordination is to have the task delay until 7.00 am and then turn the heating on. In this and other situations, this concurrency overhead is unnecessary and inefficient. Hence events and event handlers have an important role to play.

When an event occurs, it is said to be **triggered**; other terms used are **fired**, **invoked** or **delivered**. The event-handling code normally does not contain any synchronization calls that could lead to it becoming suspended; the handler runs to completion. Where these requirements cannot be guaranteed, it is necessary to schedule the event handlers rather than execute them immediately their corresponding events occur. Events can also be used as an asynchronous notification technique as described in Section 7.4.

This section focuses on time-triggered events and their handlers. Events that are triggered by the environment or other threads are also considered in Section 7.4.

10.4.1 Time-triggered events in Ada

Ada 2005 has introduced a low-level mechanism that allows a handler to be associated with a *timing event*. When the event's time is due (as determined by the real-time clock) the handler code is executed. Timing events are supported by a child package of `Ada.Real_Time` (see Program 10.6).

The handler type is an access to a protected procedure with the timing event itself been passed back to the handler when the event is triggered. The event is *set* by the attachment of a handler. Two `Set_Handler` procedures are defined, one using absolute time, the other relative. If a **null** handler is passed the event is *cleared*. This can also be achieved by calling `Cancel_Handler`. With this routine the boolean flag indicates if the event was actually set before it was cleared. If `Set_Handler` is called on an event

Program 10.6 Timing events in Ada.

```

package Ada.Real_Time.Timing_Events is
  type Timing_Event is tagged limited private;
  type Timing_Event_Handler is access protected
    procedure (Event : in out Timing_Event);
  procedure Set_Handler(Event : in out Timing_Event;
    At_Time : Time; Handler: Timing_Event_Handler);
  procedure Set_Handler(Event : in out Timing_Event;
    In_Time: Time_Span; Handler: Timing_Event_Handler);
  function Is_Handler_Set(Event : Timing_Event)
    return Boolean;
  function Current_Handler(Event : Timing_Event)
    return Timing_Event_Handler;
  procedure Cancel_Handler(Event : in out Timing_Event;
    Cancelled : out Boolean);
  function Time_Of_Event(Event : Timing_Event) return Time;
private
  -- Not specified by the language.
end Ada.Real_Time.Timing_Events;

```

that is already set then a new time is posted for the event with the original time and handler being lost. As a handler is called by a clock, it must be accessible for the lifetime of the program. Consequently, only library-level handlers can be used.

As soon as possible after the time defined for the event, the handler is executed; this clears the event. It will not be triggered again unless it is reset. The most effective way for an implementation to support timing events is to execute the handlers directly from the interrupt handler of the clock. A typical clock routine will interrupt every 10 ms or less. On each occurrence, the clock handler will check the system's delay queue to see if any tasks need to be made runnable or if any timing events need to be triggered. Tasks are moved to ready queues, handlers are executed directly. As the clock interrupt is typically the highest priority interrupt in the system, the application code's protected object (that embodies the handler procedure) must have a ceiling of `Interrupt_Priority'Last` – see Section 12.3.

Two further subprograms are defined in the support package. One allows the current handler to be obtained while the other allows the current time of the event to be requested. Both return with sensible values if the event is not set (`null` and `Ada.Real_Time.Time_First` respectively).

As the handlers for timing events are executed directly by the Ada run-time system they suffer little release jitter. Consequently, they are ideal for controlling input and output jitter. An example of this will be given in Section 10.5.

10.4.2 Time-triggered events in C/Real-Time POSIX

C/Real-Time POSIX provides support for time-triggered events through the use of signals generated from timers. Unlike Ada's timing events, both one-shot and periodic timers can be created. Furthermore, the timers can be associated with any clock supported by C/Real-Time POSIX. The interface to C/Real-Time POSIX timers is shown in Program 10.7 along with a description of the semantics of each function call.

Program 10.7 The C/Real-Time POSIX interface to timers.

```

#define TIMER_ABSTIME ..

struct itimerspec {
    struct timespec it_value; /* first timer signal */
    struct timespec it_interval; /* subsequent intervals */
};
typedef ... timer_t;

int timer_create(clockid_t clock_id, struct sigevent *evp,
                 timer_t *timerid);
    /* Create a per-process timer using the specified clock as the */
    /* timing base. evp points to a structure which contains */
    /* all the information needed concerning the signal */
    /* to be generated. */

int timer_delete(timer_t timerid);
    /* delete a per-process timer */

int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *value,
                  struct itimerspec *ovalue);
    /* Set the next expiry time for the timer specified. */
    /* If flags is set to TIMER_ABSTIME, then */
    /* the timer will expire when the clock reaches the */
    /* absolute value specified by *value.it_value */
    /* if flags is NOT set to TIMER_ABSTIME, then the timer will */
    /* expire when the interval specified by value->it_value passes */
    /* if *value.it_interval is non-zero, then a periodic timer will */
    /* go off every value->it_interval after value->it_value has */
    /* expired */
    /* Any previous timer setting is returned in *ovalue. */

int timer_gettime(timer_t timerid, struct itimerspec *value);
    /* get the details of the current timer */

int timer_getoverrun(timer_t timerid);
    /* if real-time signals are supported, return the number of signals */
    /* that have been generated by this timer but not handled */

/* All the above functions, except timer_getoverrun, return 0 if */
/* successful, otherwise -1. timer_getoverrun returns the number */
/* of overruns. When an error condition is returned by any of */
/* the above functions, a shared variable errno contains the */
/* reason for the error */

```

However, as noted in Section 7.5.1, if the process is multithreaded, the signal is sent to the whole process. Furthermore, C/Real-Time POSIX does not define the exact time the handler will be executed. Hence the only way to get predictable signal handling is to dedicate a thread to handle that signal and block it from all other threads. That thread can be given an appropriate priority and scheduled; see Chapter 11.

10.4.3 Time-triggered events in Real-Time Java

Real-Time Java also supports timers. However, unlike Ada and C/Real-Time POSIX, the resulting handlers are scheduled rather than run at the highest priority or handled in the context of an arbitrary thread. Programs 10.8 and 10.9 define the associated classes.

The abstract `Timer` class defines the base class from which timer events can be generated. All timers are based on a clock; usually the real-time clock is used. A timer has a time at which it should fire, that is, release any associated handlers. This time may be an absolute or relative time value. The appropriate release parameters associated with the timer can be created by the `createReleaseParameters` method (this overrides the method in the `AsyncEvent` class – see Section 7.7.1).

Program 10.8 The Real-Time Java Timer class.

```
package javax.realtime;
public abstract class Timer extends AsyncEvent {
    protected Timer(HighResolutionTimer time, Clock clock,
                    AsyncEventHandler handler);

    public ReleaseParameters createReleaseParameters();

    public AbsoluteTime getFireTime();
    // Get the time at which this event will fire.

    public void reschedule(HighResolutionTimer time);
    // Change the scheduled time for this event.

    public Clock getClock();

    public void disable();
    // Disable this timer, preventing it from firing. However,
    // a disabled timer continues to count while it is disabled

    public void enable();
    // Re-enable this timer after it has been disabled.
    // The timer will immediately fire if its fire
    // time has passed.

    public void start(); // start the timer ticking
    public void start(boolean disabled);
}
```

Program 10.9 The Real-Time Java OneShotTimer and PeriodicTimer classes.

```

package jaxax.realtime;
public class OneShotTimer extends Timer {
    public OneShotTimer(HighResolutionTimer time,
                        AsyncEventHandler handler);
    // Create an instance of AsyncEvent that will execute its
    // fire method at the expiration of the given time.

    public OneShotTimer(HighResolutionTimer start, Clock clock,
                        AsyncEventHandler handler);
    // Create an instance of AsyncEvent, based on the given
    // clock, that will execute its fire method at the
    // expiration of the given time.
}

public class PeriodicTimer extends Timer {

    public PeriodicTimer(HighResolutionTimer start,
                        RelativeTime interval,
                        AsyncEventHandler handler);
    // create an instance of AsyncEvent that executes its fire
    // method periodically

    public PeriodicTimer(HighResolutionTimer start,
                        RelativeTime interval,
                        Clock clock, AsyncEventHandler handler);
    // Create an instance of AsyncEvent that executes its fire
    // method periodically from start based on a particular clock

    public ReleaseParameters createReleaseParameters();

    public void setInterval(RelativeTime interval);
    // Resets the interval of this Timer
    public RelativeTime getInterval();

    public AbsoluteTime getFireTime();
}

```

Once created, a timer can be:

- **started and stopped** – indicating whether the timer should count-down or not;
- **enabled and disabled** – indicating whether the timer should fire or not;
- **rescheduled** – indicating that the timer should not fire (or not fire again) until the rescheduled time;
- **destroyed** – indicating that the resources used by the timer should be returned to the system; the timer cannot be used again.

The `getClock` and `getFireTime` methods are self-explanatory, although it should be noted that the value returned from the `getFireTime` method may return

null if the timer was created with a relative start time and has yet to be started. The `start` methods start the timer running. By default, a timer is enabled when it is started. The overloaded `start` method allows a timer to be started but disabled. Any relative time given in the constructor to a timer is converted to an absolute time at the point the `start` method is called; if an absolute time is given in the constructor, and the time has passed, the timer fires as soon as it is started (assuming it is not started disabled). Finally, the `isRunning` method returns true if the timer is both started and enabled.

Real-Time Java's asynchronous event-handling model has already been described in Section 7.7.2. An example of its use with timers to control input and output jitter is given in Section 10.5.

10.5 Controlling input and output jitter

In many application areas, particularly control systems, real-time tasks have a simple structure. They are typically periodic and read their sensor inputs at the start of each release, perform some computation, and produce output for actuators within some deadline. There are also usually some requirements on the minimum and maximum latency of the task. In these applications, significant variations in the timing of the input and output operations during each period have to be avoided. The variation is called **input and output jitter**, the latter being the variation in the actual latency of the task. In many applications, controlling jitter is performed by the I/O device itself. These 'smart' sensors and actuators can perform their I/O operations at particular points in time with minimum latency. However, where these are not available, the application must implement the necessary requirements.

Few programming real-time languages explicitly have facilities to directly specify jitter requirements. There are, however, several ways by which the programmer can use other real-time mechanism to meet the requirements. Section 9.6 showed how a jitter-constrained task could be transformed into two or three tasks each with its own deadline. Then by judicious use of scheduling parameters or temporal scopes, the required behaviour could be achieved. However, in situations where it is necessary to undertake a small computation periodically (and with minimum jitter), a time-triggered event can be used and is more efficient. These approaches will be illustrated in the following subsections.

10.5.1 Controlling I/O jitter in Ada

The input and output from and to sensors and actuators is a good example of where small amounts to computations are constrained by jitter requirements. Consider, for example, a periodic control activity. It reads a sensor, performs some computation based on the sensor value and writes a result to an actuator. In order to ensure that the environment does not become unstable, the control algorithms require that the sensor be read every 40 ms (for example); the variation in the time at which the sensor is read each cycle (input jitter) is 2 ms. The output should be written to the actuator within a deadline of 30 ms, and the output jitter no more than 4 ms. Figure 10.1 illustrates these timing constraints.

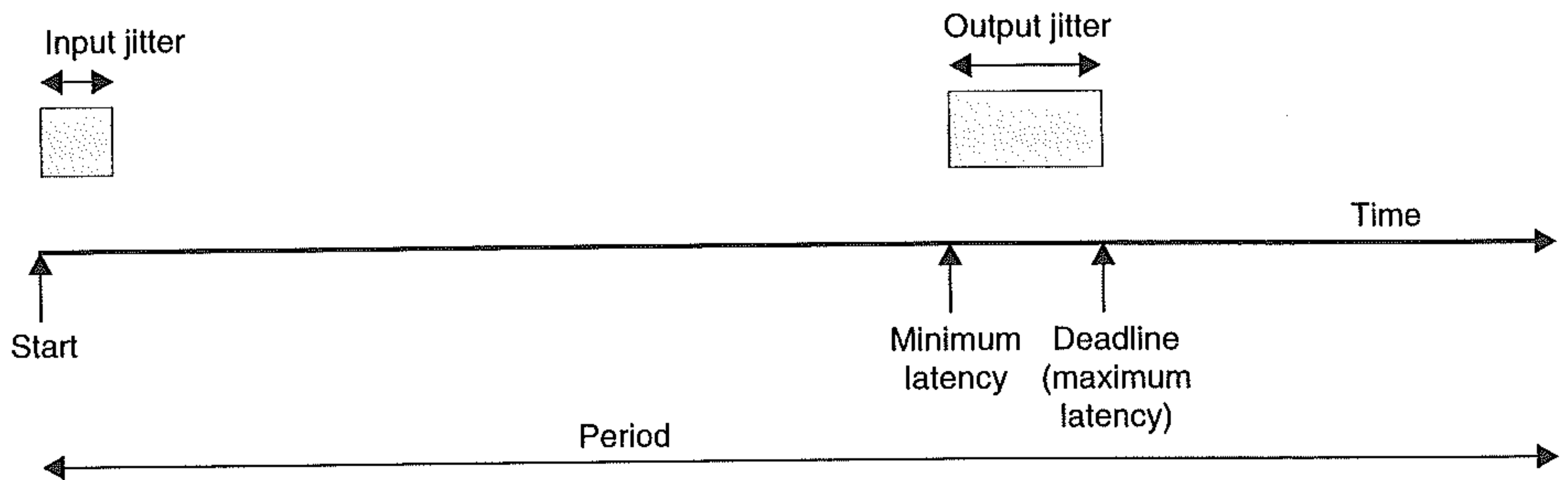


Figure 10.1 A simple task with input and output jitter constraints.

One way to get the tight time constraint in Ada is to use two timing events in conjunction with a task. First, the time constraint on the input can be implemented by the following protected type:

```
protected type Sensor_Reader is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  procedure Start;
  entry Read(Data : out Sensor_Data);
  procedure Timer(Event : in out Timing_Event);
private
  Next_Time : Time;
  Reading : Sensor_Data;
  Data_Available : Boolean := True;
end Sensor_Reader;
```

```
Input_Jitter_Control : Timing_Event;
Input_Period : Time_Span := Milliseconds(40);
```

The procedure `Start` is used to initiate the first sensor reading. The routine then sets up the next reading using the `Input_Jitter_Control` timing event. The timer will call the `Timer` procedure at the appropriate time. This will take the next sensor reading, and set up the next event. The control algorithm simply calls the `Read` entry, which becomes open every time new data is available. The body of the protected time is given below:

```
protected body Sensor_Reader is
  procedure Start is
  begin
    Reading := Read_Sensor;
    Next_Time := Clock + Input_Period;
    Data_Available := True;
    Set_Handler(Input_Jitter_Control, Next_Time, Timer'Access);
  end Start;

  entry Read(Data : out Sensor_Data) when Data_Available is
  begin
    Data := Reading;
    Data_Available := False;
  end Read;
```



```

procedure Timer(Event : in out Timing_Event) is
begin
    -- obtain Reading from sensor interface
    Data_Available := True;
    Next_Time := Next_Time + Input_Period;
    Set_Handler(Input_Jitter_Control, Next_Time, Timer'Access);
end Timer;
end Sensor_Reader;

```

The repetitive use of timing events is an effective solution for this type of requirement. A similar approach can be used for control of the output.

```

protected type Actuator_Writer is
    pragma Interrupt_Priority (Interrupt_Priority'Last);
    procedure Start;
    procedure Write(Data : Actuator_Data);
    procedure Timer(Event : in out Timing_Event);
private
    Next_Time : Time;
    Value : Actuator_Data;
end Actuator_Writer;

```

```

Output_Jitter_Control : Timing_Event;
Output_Period : Time_Span := Milliseconds(40);

```

Here, the Start routine is called 26 ms (that is, 30 – 4 ms) after starting the sensor data collection timer:

```

SR.start;
delay 0.026;
AW.start;

```

where SR and AW are instances of the two protected types.

Finally, the control algorithm task can be given. Note that it contains no 'delay until' statement. The rate is controlled by the opening and closing of the Read entry:

```

task type Control_Algorithm (Input : access Sensor_Reader;
                               Output : access Actuator_Writer);
task body Control_Algorithm is
    Input_Data : Sensor_Data;
    Output_Data : Actuator_Data;
begin
    loop
        Input.Read(Input_Data);
        -- process data;
        Output.Write(Output_Data);
    end loop;
end Control_Algorithm;

```

The use of timing events to represent a periodic activity is appropriate for small execution times or when minimum jitter on the periodic activity is required. However, it has the disadvantage that the code is always executed at interrupt priority level and hence has a temporal interference on the rest of the program. Furthermore, all input and output are treated with the same urgency. To have more control it is necessary to create

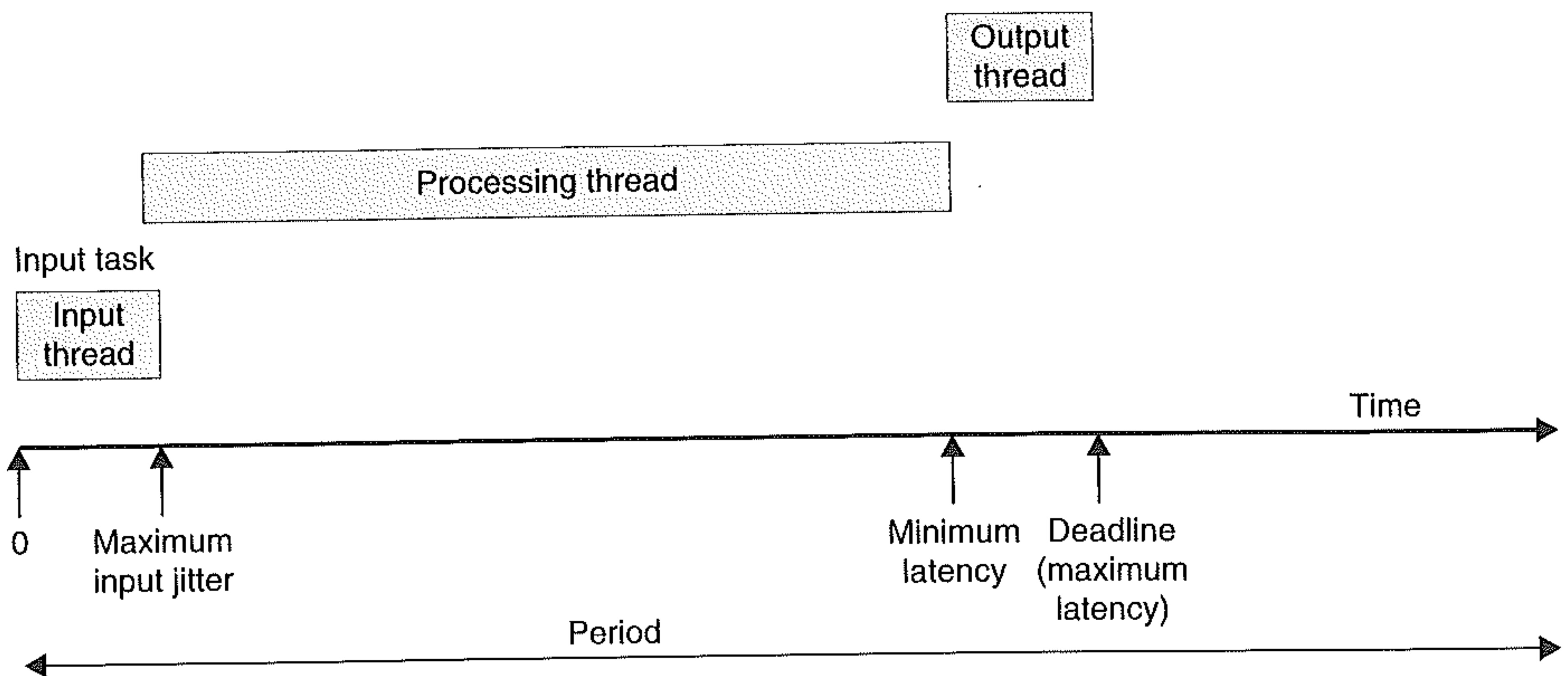


Figure 10.2 Three threads implementing input and output jitter constraints.

three tasks: one for input, one for processing and one for output. Each can be given its own deadline and scheduled so that they start at different offsets from one another. This approach is illustrated using C/Real-Time POSIX.

10.5.2 Controlling I/O jitter in C/Real-Time POSIX

It has already been mentioned in Section 10.4.2 that the precise time that signal handlers execute in response to timers is not defined. The above Ada approach could be programmed in C/Real-Time POSIX with timers using threads instead of protected objects. However, here, a slightly different solution is given. The three threads are all defined as periodic but use time offsets to ensure that they do not execute at the same time. Figure 10.2 and the following program illustrate the approach.

```
typedef struct {
    struct timespec start;
    struct timespec period;
    struct timespec max_input_jitter;
    struct timespec min_latency;
    struct timespec deadline;
} parameters;

void sensor_thread(parameters *params)
    /* destined to be the thread that reads the sensor */
{
    struct timespec next_release, remaining_time;
    /* wait until start time */
    next_release = params->start;
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                    &next_release, &remaining_time);
    while(1) {
        /* read sensor data and store in a global data -
           protected by a mutex */
        add_timespec(&next_release, &next_release, &params->period);
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                        &next_release, &remaining_time);
    }
}
```



```

    }
}

```

```

void processing_thread(parameters *params)
/* destined to be the thread that processes the sensor data*/
{
    struct timespec next_release, remaining_time;

    /* wait until first release time */
    add_timespec(&next_release, &(params->start),
                &(params->max_input_jitter));
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                    &next_release, &remaining_time);

    while(1) {
        /* get data written by input_thread,
           process data and write the value to be written to
           the actuator in global data - protected by a mutex
        */
        add_timespec(&next_release, &next_release, &params->period);
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                        &next_release, &remaining_time);
        /* code to be executed each period here */
    }
}

```

```

void actuator_thread(parameters *params)
/* destined to be the thread that write to the actuator */
{
    struct timespec next_release, remaining_time;

    /* wait until first release time */
    add_timespec(&next_release, &params->start,
                &params->min_latency);
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                    &next_release, &remaining_time);

    while(1) {
        /* get data written by processing_thread,
           process data and write the value to the actuator */
        add_timespec(&next_release, &next_release,
                    &params->period);
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                        &next_release, &remaining_time);
        /* code to be executed each period here */
    }
}

```

```

void init() {

    parameters P;
    struct timespec now;
    struct timespec init_time;
    int status;

    pthread_attr_t attributes_input;    /* thread attributes */

```

```

pthread_t PTInput;                                /* thread pointer */
pthread_attr_init(&attribute_input); /* default attributes */

pthread_create(&PTInput, &attribute_input,
              (void *) sensor_thread, &P);
// Similarly for the other threads
}

```

In the above, the `params` type contains the information needed by the threads to coordinate their actions. Each thread waits for its first release time using the `clock_nanosleep` function. It then performs its required function and calculates its next release time, and then waits until that release. Each of the above threads can have an appropriate priority set so that it meets its own time constraints (see Chapter 11).

The main disadvantage of the ‘three thread’ approach is that it requires three threads and consequently incurs the overhead that this entails. The Real-Time Java approach, given next, combines the timer and the scheduling approach.

10.5.3 Controlling I/O jitter in Real-Time Java

The Real-Time Java asynchronous event-handling model was introduced in Section 7.7.1 and the representations of timers within this model considered earlier in Section 10.4.3 of this chapter. The key goal of the Real-Time Java model is that asynchronous event handlers should be less expensive than threads. However, unlike the Ada timer event model, the timer handlers are schedulable objects.

To control input and output jitter, the sensors and actuators are read and written by asynchronous event handlers that are attached to separate periodic timers. The timers are offset by the minimum required latency. Given that the handlers are scheduled they can be given appropriate scheduling parameters. The handler reading the sensor illustrates the approach.

```

import javax.realtime.*;

public class SensorReader extends AsyncEventHandler {
    public SensorReader() {
        super();
    }

    public void handleAsyncEvent() {
        // Read sensor
        synchronized(this) {
            dataAvailable = true;
            notifyAll();
        }
    }

    public synchronized int getData() throws InterruptedException {
        while(dataAvailable == false) wait();
        dataAvailable = false;
        return inputData;
    }
}

```



```

private int inputData;
private boolean dataAvailable = false;
}

```

The processing thread is controlled in a similar manner to the Ada solution. It simply loops waiting for data, and then, once processed, passes it on to the actuator.

```

import javax.realtime.*;

public class ControlSystem extends RealtimeThread {
    public ControlSystem(SensorReader S, ActuatorWriter A) {
        super();
        SR = S;
        AW = A;
    }

    public void run() {
        while(true) {
            try {
                D = SR.getData();
                // Process data
                AW.writeData(D);
            } catch (InterruptedException ie) {}
        }
    }

    private int D;
    private SensorReader SR;
    private ActuatorWriter AW;
}

```

The main procedure configures the system (here, the setting of the priorities is not shown).

```

import javax.realtime.*;
public class Main {
    public static void main (String [] args) {

        AbsoluteTime start = Clock.getRealtimeClock().getTime();

        RelativeTime period = new RelativeTime(40,0);
        RelativeTime maxInputJitter = new RelativeTime(2,0);
        RelativeTime minLatency = new RelativeTime(26,0);

        SensorReader S = new SensorReader();

        PeriodicTimer PT1 = new PeriodicTimer(start,
            period, S);

        start = start.add(minLatency);

        ActuatorWriter A = new ActuatorWriter();
        PeriodicTimer PT2 = new PeriodicTimer(
            start,
            period,
            A);
    }
}

```

```

    ControlSystem CS = new ControlSystem(S,A);
    CS.start();
    PT1.start();
    PT2.start();
  }
}

```

The control of jitter is crucial in many application. The above three subsections have illustrated some of the approaches that can be adopted.

10.6 Other approaches for supporting temporal scopes

The previous sections have shown how mainstream real-time languages and operating systems have provided support for key temporal requirements. Of course, it is the use of these mechanisms that is the essence of real-time programming. Given the importance of this topic, it is appropriate to consider other approaches. Whilst these have not entered into the mainstream, they are used in some application areas or have some interesting ideas that have influenced the design of Ada, Real-Time Java or C/Real-Time POSIX.

10.6.1 Real-Time Euclid

Real-Time Euclid (Kligerman and Stoyenko, 1986) was one of the very first (research-oriented) languages to explore the possibility of expressing explicit temporal scopes at a process level. In Real-Time Euclid, processes are static and non-nested. Each process definition must contain activation information that pertains to its real-time behaviour (the term **frame** is used instead of temporal scope). This information takes one of two forms which relate to periodic and sporadic processes:

- (1) periodic *frameInfo* first activation *timeOrEvent*;
- (2) atEvent *conditionId* *frameInfo*.

The clause *frameInfo* defines the periodicity of the process (including the maximum rate for sporadic processes). The simplest form this can take is an expression in real-time units:

```
frame realTimeExpn
```

The value of these units is set at the beginning of the program.

A periodic process can be activated for the first time in two different ways. It can have a start time defined or it can wait for an interrupt to occur. Additionally, it can wait for either of these conditions. The syntax for *timeOrEvent* must, therefore, be one of the following:

- (1) atTime *realTimeExpn*;
- (2) atEvent *conditionId*;
- (3) atTime *realTimeExpn* or atEvent *conditionId*.

where *conditionId* is a condition variable associated with an interrupt. It is also used with sporadic processes.

To give an example of part of a Real-Time Euclid program, consider a cyclic temperature controller. Its periodicity is 60 units (that is, every minute if the time unit is set to 1 second) and it is to become active after 600 units (10 minutes) or when a *startMonitoring* interrupt arrives:

```
realTimeUnit := 1.0  % time unit = 1 seconds

var Reactor: module  % Euclid is module based
var startMonitoring : activation condition atLocation 16#A10D
% This defines a condition variable which is mapped
% onto an interrupt

    process TempController : periodic frame 60 first activation
                                atTime 600 or atEvent startMonitoring

    % import list
    %
    % execution part
    %
    end TempController
end Reactor
```

Note that there is no loop within this process. It is the scheduler that controls the required and specified periodic execution.

To illustrate how this code would need to be constructed in Ada, the process (task) part is given (a loop is needed here to force the task to cycle round):

```
task body Tempcontroller is
    -- definitions, including
    Next_Release : Duration;
begin
    select
        accept Startmonitoring;  -- or a timed entry call
                                -- onto a protected object
    or
        delay 600.0;
    end select;
    Next_Release := Clock + 60.0; -- take note of next release time
    loop
        -- execution part
        --
        delay until Next_Release;
        Next_Release := Next_Release + 60.0;
    end loop;
end Tempcontroller;
```

Not only is this more cumbersome, but the scheduler is not aware of the deadline associated with this task. Its correct execution will depend on the task becoming active again almost immediately the delay has expired.

10.6.2 Pearl

The language Pearl (Werum and Windauer, 1985) also provides explicit timing information concerning the start, frequency and termination of tasks. Unlike Real-Time Euclid, Pearl is an industrial language designed in Germany for use in process control applications.

A simple 10 second periodic task, *T*, is activated by:

```
EVERY 10 SEC ACTIVATE T
```

To activate at a particular point in time (say 12.00 noon each day):

```
AT 12:00:00 ACTIVATE LUNCH
```

A sporadic task, *S*, released by an interrupt, *IRT*, is defined by

```
WHEN IRT ACTIVATE S;
```

or if an initial delay of one second is required:

```
WHEN IRT AFTER 1 SEC ACTIVATE S;
```

Although the syntax is different, Pearl gives almost the same functionality as that described for Real-Time Euclid. The temperature controller example, however, illustrates one significant difference; a task in Pearl can be activated by a time schedule or an interrupt but *not* both. Therefore either of the following is admissible in Pearl:

```
AFTER 10 MIN ALL 60 SEC ACTIVATE TempController;
```

```
WHEN startMonitoring ALL 60 SEC ACTIVATE TempController;
```

The term `ALL 60 SEC` means repeat periodically, after the first execution, every 60 seconds.

10.6.3 DPS

Whereas Pearl and Real-Time Euclid associate temporal scopes with tasks (threads), and therefore necessitate the specification of timing constraints on the task itself, the research languages DPS (Lee and Gehlot, 1985) provide local timing facilities that apply at the block level.

In general, a DPS temporal block (scope) may need to specify three distinct timing requirements (these are similar to the more global requirements discussed earlier):

- delay start by a known amount of time;
- complete execution by a known deadline;
- take no longer than a specified time to undertake a computation.

To illustrate these structures, consider the important real-time activity of making and drinking instant coffee:

```
get_cup
put_coffee_in_cup
boil_water
put_water_in_cup
```

```
drink_coffee
replace_cup
```

The act of making a cup of coffee should take no more than 10 minutes; drinking it is more complicated. A delay of 3 minutes should ensure that the mouth is not burnt; the cup itself should be emptied within 25 minutes (it would then be cold) or before 17:00 (that is, 5 o'clock and time to go home). Two temporal scopes are required:

```
start elapse 10 do
  get_cup
  put_coffee_in_cup
  boil_water
  put_water_in_cup
end
```

```
start after 3 elapse 25 by 17:00 do
  drink_coffee
  replace_cup
end
```

For a temporal scope that is executed repetitively, a time loop construct is useful; that is,

```
from <start> to <end> every <period>
```

For example, many software engineers require regular coffee throughout the working day:

```
from 9:00 to 16:15 every 45 do
  make_and_drink_coffee
```

where `make_and_drink_coffee` could be made up of the two temporal scopes given above (minus the 'by' constraint on the drinking block). Note that if this were done, the maximum elapsed time for each iteration of the loop would be 35 minutes; this is less than the period for the loop, and therefore requires a gap between drinking two cups of coffee.

Although block-level timing constraints can be specified in this way, they result in tasks that experience different deadlines during their executions; at times they may even have no deadlines at all. By decomposing these tasks into subtasks where each subtask is a single block, it is possible to represent all deadlines as task-level constraints. The run-time scheduler is thus easier to implement; for example, in some of the algorithms discussed in the next chapter, a static priority scheme is sufficient and the scheduler does not need to be explicitly aware of deadlines.

10.6.4 Esterel

Esterel Technologies (2005) is one of a family of so-called **synchronous** languages. Like Ada, C and Java, it is an imperative language. Other languages in the family include the data-flow languages Signal (le Guernic et al., 1991) and Lustre (Halbwachs et al., 1991). Synchronous languages attempt to support verification by making certain assumptions about the temporal behaviour of their programs. The fundamental assumption underpinning this computational model is the *ideal* (or *perfect*) **synchronous hypothesis** (Berry, 1989):

Ideal systems produce their outputs synchronously with their inputs.

Hence all computation and communication are assumed to take zero time (that is, all temporal scopes are executed instantaneously). Clearly, this is a very strong, and unrealistic, assumption. However, it enables the temporal ordering of events to be determined more easily. During implementation, the *ideal synchronous hypothesis* is interpreted to imply ‘the system must execute fast enough for the effects of the synchronous hypothesis to hold’. What this means, in reality, is that following any input event, all associated outputs must occur before any new input could possibly happen. The system is then said to ‘keep up’ with its environment. In the terminology introduced in Chapter 1, Esterel is targeted at reactive systems rather than time-aware systems.

The two main components of Esterel of interest here are **modules** and **signals**. Programs are constructed from modules and are event-driven using signals for communication (which are broadcast). There are two kinds of events: *pure* and *valued*, the latter having type associated with it. Both kinds of signals are either *present* or *absent*. If present, the value of a valued signal is a data object of its type.

A very simple program is given below. There are two signals: Gas and Alarm. The former being an input valued signal that is unsigned and 8 bits, it is set by the environment. The latter is an output pure signal that will be triggered by the program.

```
module Alarm;
  input Gas : unsigned <[8]>;
  output Alarm;
  loop
    await Gas;
    if ?Gas > 50 then
      emit Alarm;
    end if;
  end loop;
end module
```

The **await** statement blocks the program until the Gas signal is present. At this point the module sets the Alarm signal via the **emit** statement if the value associated with the signal (?Gas) is greater than 50.

The following defines a periodic module that every second outputs a signal:

```
module periodic;
  output result : integer;
  var V : integer in
```



```

every Second do
  -- undertake required computation to set V
  emit ?result <= V;
end every;
end

```

One consequence of the synchronous hypothesis is that all actions are atomic. Interactions between concurrent actions are impossible as actions themselves are instantaneous. In the above example, the result is signalled at the same instant as the awaited tick (and therefore the module does not suffer from local or cumulative drift). A sporadic module that is awaiting `result` will execute ‘at the same time’ as this periodic module. This behaviour significantly reduces non-determinism. Unfortunately it also leads to potential causality problems. Consider:

```

signal S in
  present S else emit S end
end

```

This program is incoherent. If *S* is absent then it is emitted; on the other hand if it were present it would not be emitted.

A formal definition of the behavioural semantics of Esterel helps to eliminate these problems (Esterel Technologies, 2005). A program can be checked for coherence. To implement a legal Esterel program is straightforward; with the synchronous hypothesis it is always possible to construct a finite state machine. Hence a program moves from an initial state (where it reads any inputs) to a final state (where it produces any outputs). As it moves between states, no other interactions with the environment take place. As indicated earlier, as long as the finite state machine (**automaton**) is implemented with sufficient speed, the atomicity assumption can be deemed to hold.

10.6.5 Giotto

Giotto (Henzinger et al., 2001) is a domain-specific high-level research programming language for control applications. As such, its focus is on the control of input and output jitter.

The basic functional unit in Giotto is the **task**, which is a periodically executed sections of code. Several concurrent tasks make up a **mode**. Tasks can be added or removed by switching from one mode to another.

Tasks communicate with each other via typed **ports**. Ports are either *input* or *output*. The values of a task’s input ports are set when the task is released. The values of its output ports are only made available to other tasks when the task’s deadline has been reached. Hence, tasks are not allowed to communicate with each other during their execution. The code responsible for copying data from one task’s output port to another task’s input port is called a **driver**. Drivers are also responsible for reading from sensors and writing to actuators.

Driver code takes a bounded amount of execution time and is assumed to execute instantaneously. Tasks are assumed to take a non-negligible amount of time to process the data in their input ports and produce their output data. Hence, Giotto is similar to

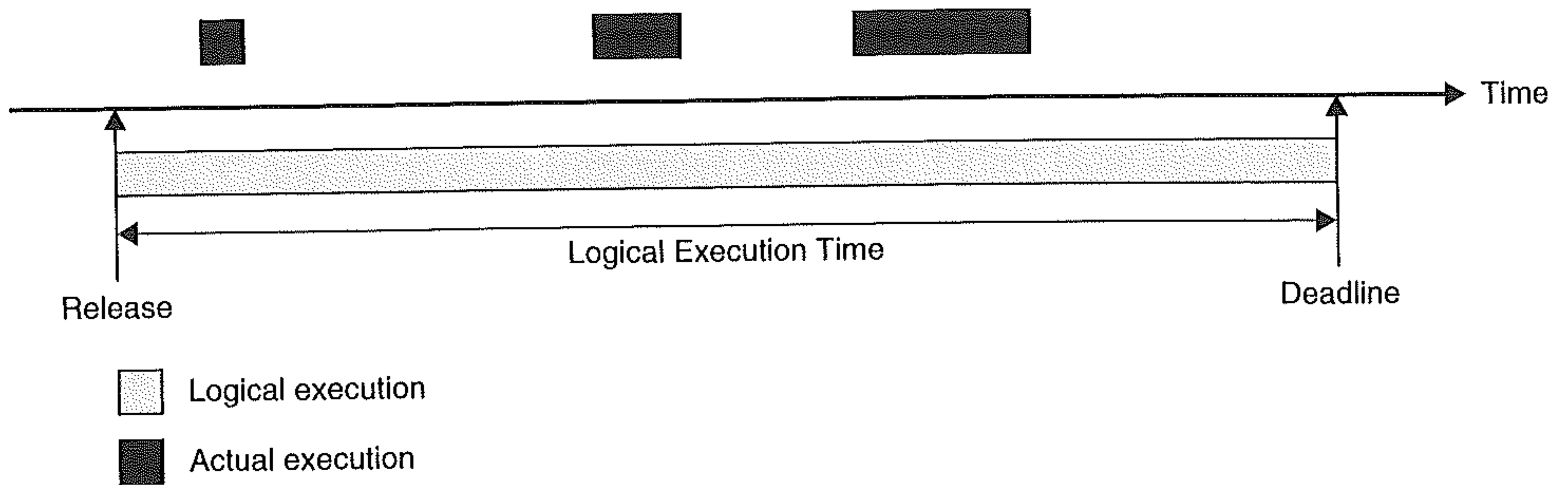


Figure 10.3 The Logical Execution Time model.

Esterel in that driver code satisfies the synchronous hypothesis. Unlike Esterel, application code does not need to support this hypothesis. Instead, the model is referred to as a **Logical Execution Time (LET)** model. This model is depicted in Figure 10.3. In terms of temporal scopes that were introduced in Section 9.6, a LET task and its associated driver code can be considered as a sampling task that consists of three temporal scopes. The first and third scopes control the input and output jitter, and are executed by the driver code.

There are various versions of Giotto; below the main characteristics of a program are shown using pseudocode. The example is part of the simple embedded system example that was introduced in Chapter 4.

```

sensor
  port temperature type integer range 10 .. 500
  port pressure type integer range 0 .. 750
actuator
  port heater type (on, off)
  port pump type integer 0 .. 9
input
  T1 type integer range 10 .. 500
  PI type integer range 0 .. 750
output
  TO type (on, off)
  PO type integer 0 .. 9

task temperature input TI output TO temperature_controller
task pressure input PI output PO pressure_controller

driver temperature_sensor
  source temperature destination TI function read_temperature
driver pressure_sensor
  source pressure destination PI function read_pressure
driver heater_actuator
  source TO destination heater function write_heater
driver pump_actuator
  source PO destination pump function write_pump

mode normal period 20 ports TO, PO
  frequency 2 invoke temperature driver temperature_sensor

```



```

frequency 1 invoke pressure_driver pressure_sensor
frequency 2 update heater_actuator
frequency 1 update pump_actuator
start normal

```

Giotto is more concerned with the architecture of the program rather than the details of a task's or driver's code. It focuses on the declaration of ports (sensors, actuators, input and output), tasks and drivers. The above code shows that there are two sensors and two actuators and the type of their associated data. Associated with each sensor is a driver that is responsible for taking the data from the devices' registers and placing the values in the input ports of the two tasks. Similarly, drivers are responsible for taking the data from the output ports of the two tasks and writing the values to the actuators' device registers.

The scheduling of the system is specified using the mode construct. In this simple example, a single mode is defined (called `normal`). The system is time triggered with a period of 20 ms. The schedule consists of the frequencies of the two tasks within this period – hence task `temperature` runs every 10 ms, `pressure` every 20 ms. The tasks' inputs are associated with the appropriate drivers. Similarly, the actuators have associated drivers. The semantics of the statement are that the drivers associated with the `invoke` statement must be scheduled at the release time of the task according to the synchronous hypothesis. The drivers associated with the `update` statement are scheduled at the deadlines of the tasks (here equal to their periods).

Giotto is a good example of a language targeted to the characteristics of a particular application domain. Its restrictive communication model, however, means that it is not appropriate for systems where more dynamic interactions between tasks are required.

Summary

The provision of abstractions that match the typical requirements found in an application domain makes an enormous difference to the ease with which a language can be used in that domain. However, there is a trade-off. Too many domain-specific programming languages lead to development problems, as the more esoteric the language the more difficult it is to find programmers. Languages like Ada and Java walk a tightrope trying to be generic and yet relevant to the real-time and embedded systems domains. The C/Real-Time POSIX standard attempts to augment POSIX to address similar concerns. Languages like Esterel and Giotto show the advantages of providing a restrictive computation model. Of course, these computational models can also be implemented in general-purpose real-time languages if the appropriate restrictions are adhered to (and perhaps enforced by analysis tools).

This chapter has introduced some of the common abstractions for representing typical temporal scopes, and shown how they are explicitly supported or how they can be programmed. In particular, the ability to:

- program periodic/sporadic and aperiodic tasks, and
- control input and latency (output) jitter

has been examined. Ada and C/Real-Time POSIX support primitive mechanisms that allow the above to be programmed. Real-Time Java provides higher-level abstractions.

Further reading

- Caspi, P., Raymond, P. and Tripakis, S. (2007) Synchronous programming, in *Handbook of Real-Time and Embedded Systems*, I. Lee, J. Y-T. Leung and S. M. Son (eds). New York: Chapman and Hall.
- Dibble, P. C. (2008) *Real-Time Java Platform Programming*, 2nd edn, Book Surge Publishing, www.booksurge.com.
- Kirsch, C. M. and Sengupta, R. (2007) *The Evolution of Real-Time Programming*, Handbook of Real-Time and Embedded Systems. New York: Chapman and Hall.

Exercises

- 10.1 Show how the Real-Time Java `waitForNextPeriod` method can be implemented in Ada.
- 10.2 To what extent can a combination of Ada's timing event and protected objects allow the Giotto input and output ports to be implemented?
- 10.3 To what extent can a combination of Real-Time Java's timers, asynchronous event handlers and synchronized objects allow the Giotto input and output ports to be implemented?
- 10.4 To what extent can a combination of C/Real-Time POSIX's timers and mutexes allow the Giotto input and output ports to be implemented?

Chapter 11

Scheduling real-time systems

11.1	The cyclic executive approach	11.10	An extendible task model for FPS
11.2	Task-based scheduling	11.11	Earliest deadline first (EDF) scheduling
11.3	Fixed-priority scheduling (FPS)	11.12	Dynamic systems and online analysis
11.4	Utilization-based schedulability tests for FPS	11.13	Worst-case execution time
11.5	Response time analysis (RTA) for FPS	11.14	Multiprocessor scheduling
11.6	Sporadic and aperiodic tasks	11.15	Scheduling for power-aware systems
11.7	Task systems with $D < T$	11.16	Incorporating system overheads
11.8	Task interactions and blocking		Summary
11.9	Priority ceiling protocols		Further reading
			Exercises

In a concurrent program, it is not necessary to specify the exact order in which tasks execute. Synchronization primitives are used to enforce the local ordering constraints, such as mutual exclusion, but the general behaviour of the program exhibits significant non-determinism. If the program is correct then its functional outputs will be the same regardless of internal behaviour or implementation details. For example, five independent tasks can be executed non-preemptively in 120 different ways on a single processor. With a multiprocessor system or preemptive behaviour, there are infinitely more interleavings.

While the program's outputs will be identical with all these possible interleavings, the timing behaviour will vary considerably. If one of the five tasks has a tight deadline then perhaps only interleavings in which it is executed first will meet the program's temporal requirements. A real-time system needs to restrict the non-determinism found within concurrent systems. This activity is known as scheduling. In general, a scheduling scheme provides two features:

- An algorithm for ordering the use of system resources (in particular the CPUs).
- A means of predicting the worst-case behaviour of the system when the scheduling algorithm is applied.

The predictions can then be used to confirm that the temporal requirements of the system are satisfied.

A scheduling scheme can be **static** (if the predictions are undertaken before execution) or **dynamic** (if run-time decisions are used). This chapter will concentrate mainly on static schemes. Most attention will be given to preemptive priority-based schemes on a single processor system. Here, tasks are assigned priorities such that at all times the task with the highest priority is executing (if it is not delayed or otherwise suspended). A scheduling scheme will therefore involve a priority assignment algorithm and a schedulability test. Other scheduling approaches, such as EDF, and multiprocessor and energy issues are also covered in this chapter. The first approach to be review, however, will be the traditional scheme involving the production of a cyclic executive. All issues concerned with programming schedulable systems are covered in the next chapter.

11.1 The cyclic executive approach

With a fixed set of purely periodic tasks, it is possible to lay out a complete schedule such that the repeated execution of this schedule will cause all tasks to run at their correct rate. The cyclic executive is, essentially, a table of procedure calls, where each procedure represents part of the code for a ‘task’. The complete table is known as the **major cycle**; it typically consists of a number of **minor cycles** each of fixed duration. So, for example, four minor cycles of 25 ms duration would make up a 100 ms major cycle. During execution, a clock interrupt every 25 ms will enable the scheduler to loop through the four minor cycles. Table 11.1 provides a task set that must be implemented via a simple four-slot major cycle. A possible mapping onto the cyclic executive is shown in Figure 11.1, which illustrates the job that the processor is executing at any particular time.

Task	Period, T	Computation time, C
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

Table 11.1 Cyclic executive task set.

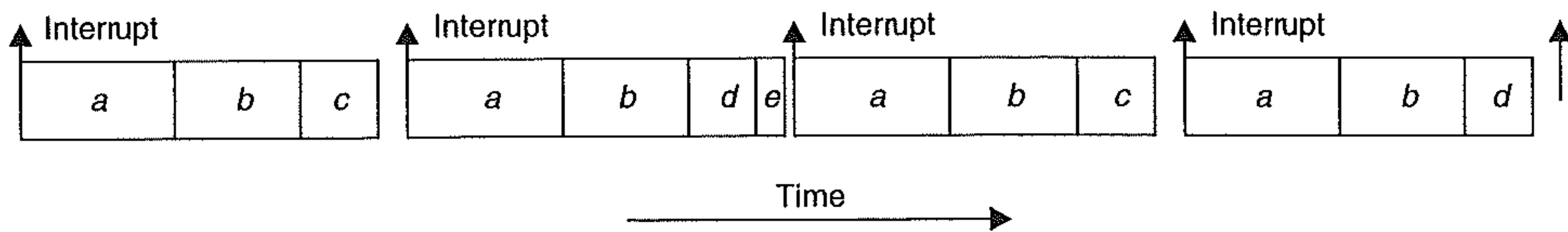


Figure 11.1 Time-line for task set.

Even this simple example illustrates some important features of this approach.

- No actual tasks exist at run-time; each minor cycle is just a sequence of procedure calls.
- The procedures share a common address space and can thus pass data between themselves. This data does not need to be protected (via a semaphore, for example) because concurrent access is not possible.
- All 'task' periods must be a multiple of the minor cycle time.

This final property represents one of the major drawbacks of the cyclic executive approach; others include (Locke, 1992):

- the difficulty of incorporating sporadic tasks;
- the difficulty of incorporating tasks with long periods; the major cycle time is the maximum period that can be accommodated without secondary schedules (that is, a procedure in a major cycle that will call a secondary procedure every N major cycles);
- the difficulty of actually constructing the cyclic executive;
- any 'task' with a sizeable computation time will need to be split into a fixed number of fixed sized procedures (this may cut across the structure of the code from a software engineering perspective, and hence may be error-prone);

If it is possible to construct a cyclic executive then no further schedulability test is needed (the scheme is 'proof by construction'). However, for systems with high utilization, the building of the executive is problematic. An analogy with the classical bin packing problem can be made. With that problem, items of varying sizes (in just one dimension) have to be placed in the minimum number of bins such that no bin is over-full. The bin packing problem is known to be NP-hard and hence is computationally infeasible for sizeable problems (a typical realistic system will contain perhaps 40 minor cycles and 400 entries). Heuristic sub-optimal schemes must therefore be used.

Although for simple periodic systems, the cyclic executive will remain an appropriate implementation strategy, a more flexible and accommodating approach is furnished by the task-based scheduling schemes. These approaches will therefore be the focus in the remainder of this chapter.

11.2 Task-based scheduling

With the cyclic executive approach, at run-time, only a sequence of procedure calls is executed. The notion of task (thread) is not preserved during execution. An alternative approach is to support task execution directly (as is the norm in general-purpose operating systems) and to determine which task should execute at any one time by the use of one or more scheduling attributes. With this approach, a task is deemed to be in one of a number of *states* (assuming no intertask communication):

- runnable;
- suspended waiting for a timing event – appropriate for periodic tasks;
- suspended waiting for a non-timing event – appropriate for sporadic tasks.

11.2.1 Scheduling approaches

There are, in general, a large number of different scheduling approaches. In this book we will consider three.

- **Fixed-Priority Scheduling (FPS)** – this is the most widely used approach and is the main focus of this chapter. Each task has a fixed, **static**, priority which is computed pre-run-time. The runnable tasks are executed in the order determined by their priority. *In real-time systems, the ‘priority’ of a task is derived from its temporal requirements, not its importance to the correct functioning of the system or its integrity.*
- **Earliest Deadline First (EDF) Scheduling** – here the runnable tasks are executed in the order determined by the absolute deadlines of the tasks; the next task to run being the one with the shortest (nearest) deadline. Although it is usual to know the relative deadlines of each task (e.g. 25 ms after release), the absolute deadlines are computed at run-time, and hence the scheme is described as **dynamic**.
- **Value-Based Scheduling (VBS)** – if a system can become overloaded (current utilization greater than 100%) then the use of simple static priorities or deadlines is not sufficient; a more **adaptive** scheme is needed. This often takes the form of assigning a *value* to each task and employing an online value-based scheduling algorithm to decide which task to run next.

As indicated earlier, the bulk of this chapter is concerned with FPS as it is supported by various real-time languages and operating system standards. The use of EDF is also important and some consideration of its analytical basis is given in the following discussions. A short description of the use of VBS is given towards the end of the chapter in Section 11.12.

11.2.2 Scheduling characteristics

There are a number of important characteristics that can be ascribed to a scheduling test. The two most important are **sufficiency** and **necessity**.

- A schedulability test is defined to be **sufficient** if a positive outcome guarantees that all deadlines are always met.
- A test can also be labelled as **necessary** if failure of the test will indeed lead to a deadline miss at some point during the execution of the system.

A **sufficient and necessary** test is **exact** and hence is in some sense optimal; a sufficient but not necessary test is pessimistic, but for many situations an exact test is intractable. From an engineering point of view, a tractable sufficient test with low pessimism is ideal.

A scheduling test is usually applied to the worst-case behavioural description of the application. A system is schedulable with respect to a specified scheduling policy if it will meet all its timing requirements when executed on its target platform with that

scheduling policy. A scheduling test is said to be **sustainable** if it correctly predicts that a schedulable system will remain schedulable when its operational parameters ‘improve’ – for example, if a system is schedulable it should remain so if some of its tasks have their periods or deadlines increased, or their resource requirement reduced; or if the application is moved to a faster processor.

11.2.3 Preemption and non-preemption

With priority-based scheduling, a high-priority task may be released during the execution of a lower-priority one. In a **preemptive** scheme, there will be an immediate switch to the higher-priority task. Alternatively, with **non-preemption**, the lower-priority task will be allowed to complete before the other executes. In general, preemptive schemes enable higher-priority tasks to be more reactive, and hence they are preferred. Between the extremes of preemption and non-preemption, there are alternative strategies that allow a lower-priority task to continue to execute for a bounded time (but not necessarily to completion). These schemes are known as **deferred preemption** or **cooperative dispatching**. These will be considered again in Section 11.10.3. Before then, dispatching will be assumed to be preemptive. Schemes such as EDF and VBS can also take on a preemptive or non-preemptive form.

11.2.4 Simple task model

An arbitrarily complex concurrent program cannot easily be analysed to predict its worst-case behaviour. Hence it is necessary to impose some restrictions on the structure of real-time concurrent programs. This section will present a very simple model in order to describe some standard scheduling schemes. The model is generalized in later sections of this chapter. The basic model has the following characteristics.

- The application is assumed to consist of a fixed set of tasks.
- All tasks are periodic, with known periods.
- The tasks are completely independent of each other.
- All system overheads, context-switching times and so on are ignored (that is, assumed to have zero cost).
- All tasks have deadlines equal to their periods (that is, each task must complete before it is next released).¹
- All tasks have fixed worst-case execution times.
- No task contains any internal suspension points (e.g. an internal delay statement or a blocking I/O request).
- All tasks execute on a single processor (CPU).

¹As the deadline is derived from the task’s period it is sometimes referred to as an *implicit* deadline. If the deadline value is different from the period then the deadline is *explicit*.

Notation	Description
B	Worst-case blocking time for the task (if applicable)
C	Worst-case execution time (WCET) of the task
D	Deadline of the task
I	The interference time of the task
J	Release jitter of the task
N	Number of tasks in the system
P	Priority assigned to the task (if applicable)
R	Worst-case response time of the task
T	Minimum time between task releases (task period)
U	The utilization of each task (equal to C/T)
$a - z$	The name of a task

Table 11.2 Standard notation.

One consequence of the task’s independence is that it can be assumed that at some point in time all tasks will be released together. This represents the maximum load on the processor and is known as a **critical instant**. Table 11.2 gives a standard set of notations for task characteristics.

Each task is assumed to give rise to a (potentially) infinite series of executions. Each execution is known as an **invocation** (release) of the task or simply as a **job**.

11.3 Fixed-priority scheduling (FPS)

With the straightforward model outlined above, there exists a simple optimal priority assignment scheme for FPS known as **rate monotonic** priority assignment. Each task is assigned a (unique) priority based on its period: the shorter the period, the higher the priority (that is, for two tasks i and j , $T_i < T_j \Rightarrow P_i > P_j$). This assignment is optimal in the sense that if any task set can be scheduled (using preemptive priority-based scheduling) with a fixed-priority assignment scheme, then the given task set can also be scheduled with a rate monotonic assignment scheme. Table 11.3 illustrates a five task set and shows what the relative priorities must be for optimal temporal behaviour. Note that priorities are represented by integers, and that the higher the integer, the greater the priority. Care must be taken when reading other books and papers on priority-based

Task	Period, T	Priority, P
a	25	5
b	60	3
c	42	4
d	105	1
e	75	2

Table 11.3 Example of priority assignment.

scheduling, as often priorities are ordered the other way; that is, priority 1 is the highest. In this book, *priority 1 is the lowest*, as this is the normal usage in most programming languages and operating systems.

11.4 Utilization-based schedulability tests for FPS

This section describes a very simple schedulability test for FPS which, although not exact, is attractive because of its simplicity.

Liu and Layland (1973) showed that by considering only the utilization of the task set, a test for schedulability can be obtained (when the rate monotonic priority ordering is used). If the following condition is true then all N tasks will meet their deadlines (note that the summation calculates the total utilization of the task set):

$$\sum_{i=1}^N \left(\frac{C_i}{T_i} \right) \leq N(2^{1/N} - 1)$$

(11.1)

Table 11.4 shows the utilization bound (as a percentage) for small values of N . For large N , the bound asymptotically approaches 69.3%. Hence any task set with a combined utilization of less than 69.3% will always be schedulable by a preemptive priority-based scheduling scheme, with priorities assigned by the rate monotonic algorithm.

Three simple examples will now be given to illustrate the use of this test. In these examples, the units (absolute magnitudes) of the time values are not defined. As long as all the values (T s, C s and so on) are in the same units, the tests can be applied. So in these (and later examples), the unit of time is just considered to be a *tick* of some notional time base.

Table 11.5 contains three tasks that have been allocated priorities via the rate monotonic algorithm (hence task c has the highest priority and task a the lowest). Their

N	Utilization bound
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%

Table 11.4 Utilization bounds.

Task	Period, T	Computation time, C	Priority, P	Utilization, U
a	50	12	1	0.24
b	40	10	2	0.25
c	30	10	3	0.33

Table 11.5 Task set A.

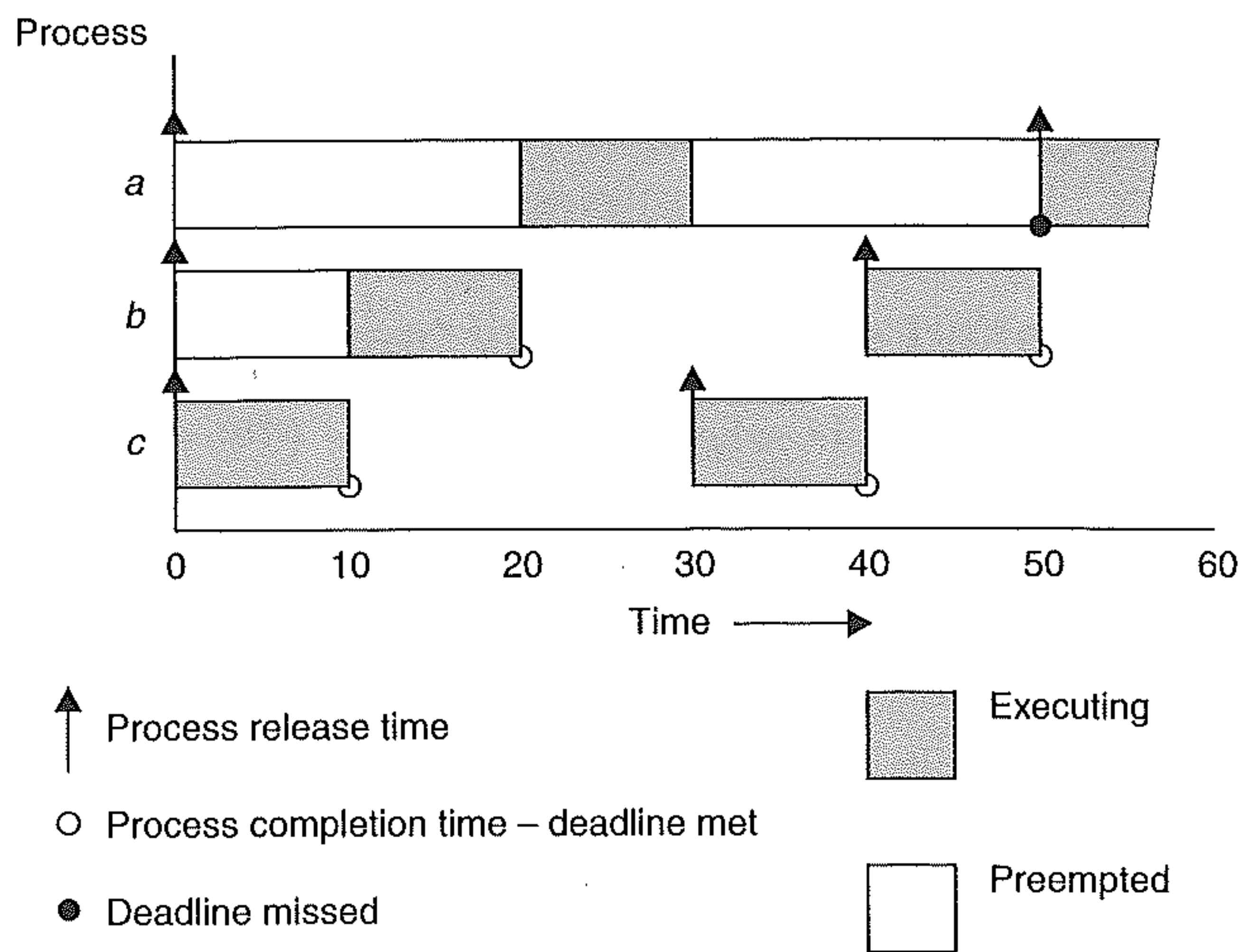


Figure 11.2 Time-line for task set A.

combined utilization is 0.82 (or 82%). This is above the threshold for three tasks (0.78), and hence this task set fails the utilization test.

The actual behaviour of this task set can be illustrated by drawing out a **time-line**. Figure 11.2 shows how the three tasks would execute if they all started their executions at time 0. Note that, at time 50, task *a* has consumed only 10 ticks of execution, whereas it needed 12, and hence it has missed its first deadline.

Time-lines are a useful way of illustrating execution patterns. For illustration, Figure 11.2 is drawn as a **Gantt chart** in Figure 11.3.

The second example is contained in Table 11.6. Now the combined utilization is 0.775, which is below the bound, and hence this task set is guaranteed to meet all its deadlines. If a time-line for this set is drawn, all deadlines would be satisfied.

Although cumbersome, time-lines can actually be used to test for schedulability. But how far must the line be drawn before one can conclude that the future holds no surprises? For task sets that share a common release time (that is, they share a *critical instant*), it can be shown that a time-line equal to the size of the longest period is sufficient (Liu and Layland, (1973)). So if all tasks meet their first deadline then they will meet all future ones.

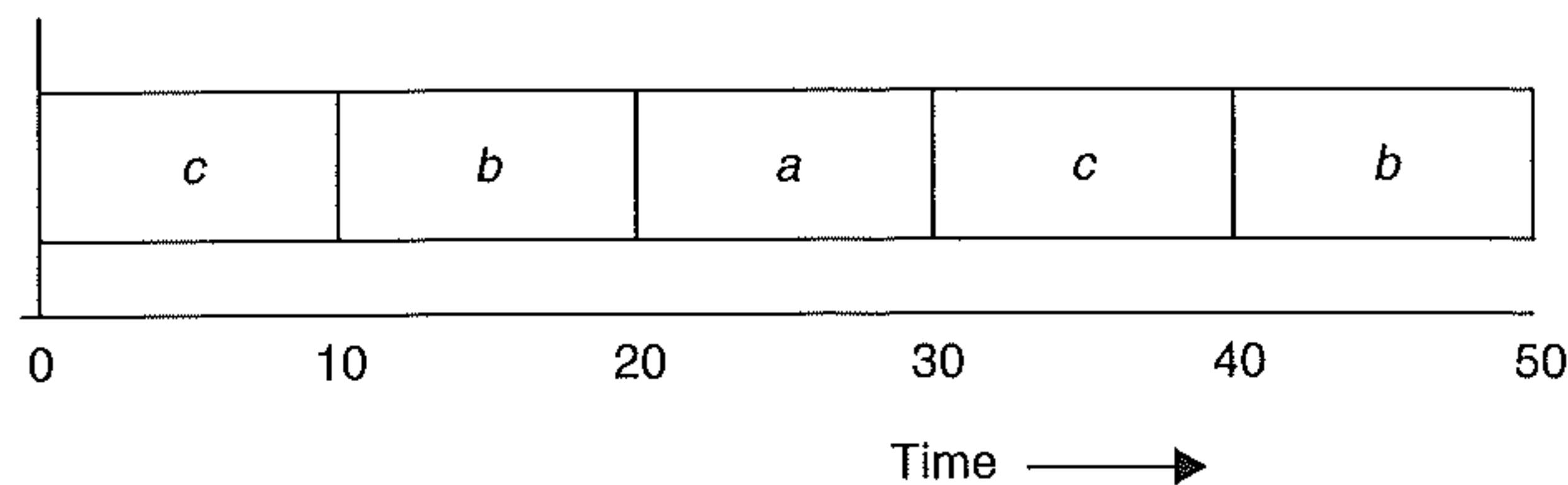


Figure 11.3 Gantt chart for task set A.

Task	Period, T	Computation time, C	Priority, P	Utilization, U
a	80	32	1	0.400
b	40	5	2	0.125
c	16	4	3	0.250

Table 11.6 Task set B.

Task	Period, T	Computation time, C	Priority, P	Utilization, U
a	80	40	1	0.50
b	40	10	2	0.25
c	20	5	3	0.25

Table 11.7 Task set C.

A final example is given in Table 11.7. This is again a three-task system, but the combined utility is now 100%, so it clearly fails the test. At run-time, however, the behaviour seems correct, all deadlines are met up to time 80 (see Figure 11.4). Hence the task set fails the test, but at run-time does not miss a deadline. Therefore, the test is *sufficient* but not *necessary*. If a task set passes the test, it *will* meet all deadlines; if it fails the test, it *may* or *may not* fail at run-time. A final point to note about this utilization-based test is that it only supplies a simple yes/no answer. It does not give any indication of the actual response times of the tasks. This is remedied in the response time approach described in Section 11.5.

11.4.1 Improved utilization-based tests for FPS

Since the publication of the Lui and Layland utilization bound a number of improvements have been developed. Here two alternative schemes are considered. First a simple re-interpretation of Equation (11.1) can be employed. Rather than the N standing for the

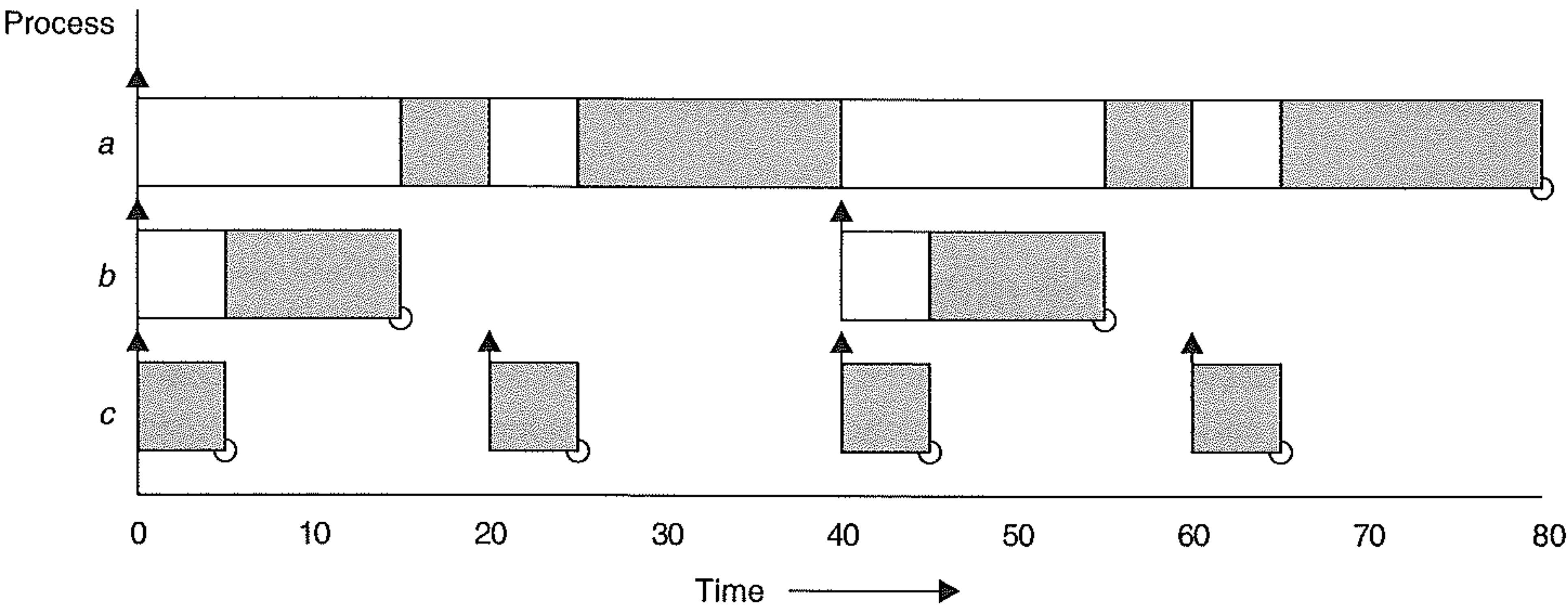


Figure 11.4 Time-line for task set C.

number of tasks, it can be defined to be the number of distinct task families in the application. A family of tasks have periods that are multiples of a common value (for example 8, 16, 64 and 128).

Consider the task sets defined earlier. For task set B (Table 11.6) there are three tasks but only two families (the 80 and 40 periods imply a single family). So the bound for this system is now 0.828 (not 0.78). The utilization of task set B is 0.775 so is below both bounds. However, if the period of task *c* is shortened to 14 (from 16) then the utilization of the task set rises to 0.81 (approximately) – this is above the Lui and Layland bound but below the new bound and hence this new task set is correctly deemed schedulable by this new test.

For task set C (see Table 11.7) there is an even more impressive improvement. Now there is only one family (as the periods are 80, 40 and 20). So the utilization bound is 1.0 and hence this system is schedulable by this test. Although this result shows the effectiveness of this approach there is a drawback with this test – it is not sustainable. Consider a minor change to the characteristics of this task set; let the period of task *a* move from 80 to 81. This alteration should make the system easier to schedule; a period has been extended and hence the overall utilization has been reduced (though only by a small amount from 1 to 0.994). But the move from 80 to 81 results in there now being two families and not just one, so the bound drops from 1 to 0.82. The new system cannot be proven to be schedulable (although it clearly is if the original task set was schedulable).

Another improved utilization-based test was developed by Bini et al. (2007) and has a different form:

$$\prod_{i=1}^N \left(\frac{C_i}{T_i} + 1 \right) \leq 2 \quad (11.2)$$

To give a simple example of the use of this formulation, consider again task set B (Table 11.6) with the minor modification that the period of task *a* is now 76 (rather than 80). The total utilization of this new system is .796 which is above the bound for three tasks, and hence schedulability is unproven. Note there are now three families so no improvement from the other approach. Applying Equation (11.2)

$$1.421 * 1.125 * 1.25 = 1.998 < 2$$

indicates that the system is schedulable by this test and, indeed, a time-line for this revised task set would show that all deadlines have been met.

11.5 Response time analysis (RTA) for FPS

The utilization-based tests for FPS have two significant drawbacks: they are not exact, and they are not really applicable to a more general task model. This section provides a different form of test. The test is in two stages. First, an analytical approach is used to predict the worst-case response time (*R*) of each task. These values are then compared, trivially, with the task deadlines. This requires each task to be analysed individually.

For the highest-priority task, its worst-case response time will equal its own computation time (that is, $R = C$). Other tasks will suffer **interference** from higher-priority

tasks; this is the time spent executing higher-priority tasks when a low-priority task is runnable. So for a general task i :

$$R_i = C_i + I_i \quad (11.3)$$

where I_i is the maximum interference that task i can experience in any time interval $(t, t + R_i)$.² The maximum interference obviously occurs when all higher-priority tasks are released at the same time as task i (that is, at a critical instant). Without loss of generality, it can be assumed that all tasks are released at time 0. Consider one task (j) of higher priority than i . Within the interval $[0, R_i)$, it will be released a number of times (at least one). A simple expression for this number of releases is obtained using a ceiling function:

$$\text{Number_Of_Releases} = \left\lceil \frac{R_i}{T_j} \right\rceil$$

The ceiling function ($\lceil \cdot \rceil$) gives the smallest integer greater than the fractional number on which it acts. So the ceiling of $1/3$ is 1, of $6/5$ is 2, and of $6/3$ is 2. The definitions of the ceilings of negative values need not be considered. Later in this chapter floor functions are employed; they compute the largest integer smaller than the fractional part meaning that the floor of $1/3$ is 0, of $6/5$ is 1 and of $6/3$ is again 2.

So, if R_i is 15 and T_j is 6 then there are three releases of task j (at times 0, 6 and 12). Each release of task j will impose an interference of C_j . Hence:

$$\text{Maximum_Interference} = \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

If $C_j = 2$ then in the interval $[0, 15)$ there are 6 units of interference. Each task of higher priority is interfering with task i , and hence:

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

where $hp(i)$ is the set of higher-priority tasks (than i). Substituting this value back into Equation (11.3) gives (Joseph and Pandya, 1986):

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad (11.4)$$

Although the formulation of the interference equation is exact, the actual amount of interference is unknown as R_i is unknown (it is the value being calculated). Equation (11.4) has R_i on both sides, but is difficult to solve due to the ceiling functions. It is actually an example of a fixed-point equation. In general, there will be many values of R_i that form solutions to Equation (11.4). The smallest such value of R_i represents the worst-case response time for the task. The simplest way of solving Equation (11.4) is to

²Note that as a discrete time model is used in this analysis, all time intervals must be closed at the beginning (denoted by '[') and open at the end (denoted by a ')'). Thus a task can complete executing on the same tick as a higher-priority task is released.

form a recurrence relationship (Audsley et al., 1993a):

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \quad (11.5)$$

The set of values $\{w_i^0, w_i^1, w_i^2, \dots, w_i^n, \dots\}$ is, clearly, monotonically non-decreasing. When $w_i^n = w_i^{n+1}$, the solution to the equation has been found. If $w_i^0 < R_i$ then w_i^n is the smallest solution and hence is the value required. If the equation does not have a solution then the w values will continue to rise (this will occur for a low-priority task if the full set has a utilization greater than 100%). Once they get bigger than the task's period, T , it can be assumed that the task will not meet its deadline. The starting value for the process, w_i^0 , must not be greater than the final (unknown) solution R_i . As $R_i \geq C_i$ a safe starting point is C_i – there are, however, more efficient starting values (Davis et al., 2008).

The above analysis gives rise to the following algorithm for calculation response times:

```

for i in 1..N loop -- for each task in turn
  n := 0
  w_i^n := C_i
  loop
    calculate new w_i^{n+1} from Equation (11.5)
    if w_i^{n+1} = w_i^n then
      R_i := w_i^n
      exit value found
    end if
    if w_i^{n+1} > T_i then
      exit value not found
    end if
    n := n + 1
  end loop
end loop

```

By implication, if a response time is found it will be less than T_i , and hence less than D_i , its deadline (remember with the simple task model $D_i = T_i$).

In the above discussion, w_i has been used merely as a mathematical entity for solving a fixed-point equation. It is, however, possible to get an intuition for w_i from the problem domain. Consider the point of release of task i . From that point, until the task completes, the processor will be executing tasks with priority P_i or higher. The processor is said to be executing a **P_i -busy period**. Consider w_i to be a time window that is moving down the busy period. At time 0 (the notional release time of task i), all higher-priority tasks are assumed to have also been released, and hence:

$$w_i^1 = C_i + \sum_{j \in hp(i)} C_j$$

This will be the end of the busy period unless some higher-priority task is released a second time. If it is, then the window will need to be pushed out further. This continues with the window expanding and, as a result, more computation time falling into the window. If this continues indefinitely then the busy period is unbounded (that is, there

Task	Period, T	Computation time, C	Priority, P
a	7	3	3
b	12	3	2
c	20	5	1

Table 11.8 Task set D.

is no solution). However, if at any point, an expanding window does not suffer an extra ‘hit’ from a higher-priority task then the busy period has been completed, and the size of the busy period is the response time of the task.

To illustrate how the RTA is used, consider task set D given in Table 11.8. The highest-priority task, a , will have a response time equal to its computation time (for example, $R_a = 3$). The next task will need to have its response time calculated. Let w_b^0 equal the computation time of task b , which is 3. Equation (11.5) is used to derive the next value of w :

$$w_b^1 = 3 + \left\lceil \frac{3}{7} \right\rceil 3$$

that is, $w_b^1 = 6$. This value now balances the equation ($w_b^2 = w_b^1 = 6$) and the response time of task b has been found (that is, $R_b = 6$).

The final task will give rise to the following calculations:

$$w_c^0 = 5$$

$$w_c^1 = 5 + \left\lceil \frac{5}{7} \right\rceil 3 + \left\lceil \frac{5}{12} \right\rceil 3 = 11$$

$$w_c^2 = 5 + \left\lceil \frac{11}{7} \right\rceil 3 + \left\lceil \frac{11}{12} \right\rceil 3 = 14$$

$$w_c^3 = 5 + \left\lceil \frac{14}{7} \right\rceil 3 + \left\lceil \frac{14}{12} \right\rceil 3 = 17$$

$$w_c^4 = 5 + \left\lceil \frac{17}{7} \right\rceil 3 + \left\lceil \frac{17}{12} \right\rceil 3 = 20$$

$$w_c^5 = 5 + \left\lceil \frac{20}{7} \right\rceil 3 + \left\lceil \frac{20}{12} \right\rceil 3 = 20$$

Hence R_c has a worst-case response time of 20, which means that it will just meet its deadline. This behaviour is illustrated in the Gantt chart shown in Figure 11.5.

Consider again the task set C. This set failed the utilization-based test but was observed to meet all its deadlines up to time 80. Table 11.9 shows the response times calculated by the above method for this collection. Note that all tasks are now predicted to complete before their deadlines.

The response time calculations have the advantage that they are sufficient and necessary – if the task set passes the test they will meet all their deadlines; if they fail

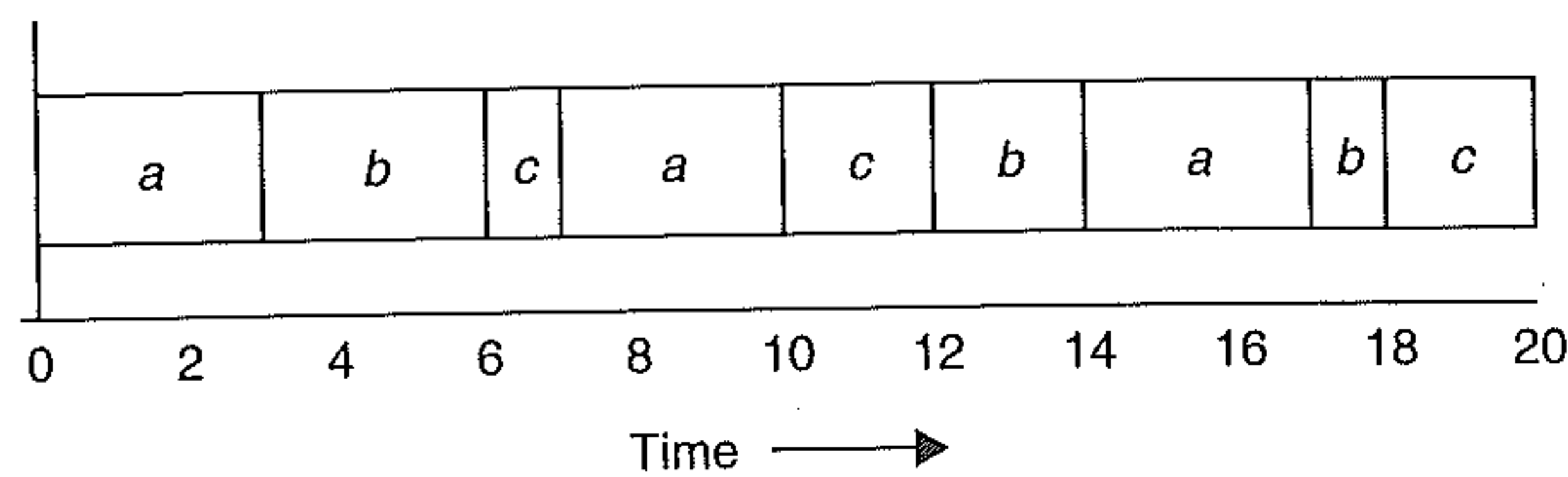


Figure 11.5 Gantt chart for task set D.

Task	Period, T	Computation time, C	Priority, P	Response time, R
a	80	40	1	80
b	40	10	2	15
c	20	5	3	5

Table 11.9 Response time for task set C.

the test, then, at run-time, a task will miss its deadline (unless the computation time estimations, C , themselves turn out to be pessimistic). As these tests are superior to the utilization-based ones, this chapter will concentrate on extending the applicability of the response time method.

11.6 Sporadic and aperiodic tasks

To expand the simple model of Section 11.2.4 to include sporadic (and aperiodic) task requirements, the value T is interpreted as the minimum (or average) inter-arrival interval (Audsley et al., 1993a). A sporadic task with a T value of 20 ms is guaranteed not to arrive more than once in any 20 ms interval. In reality, it may arrive much less frequently than once every 20 ms, but the response time test will ensure that the maximum rate can be sustained (if the test is passed!).

The other requirement that the inclusion of sporadic tasks demands concerns the definition of the deadline. The simple model assumes that $D = T$. For sporadic tasks, this is unreasonable. Often a sporadic is used to encapsulate an error-handling routine or to respond to a warning signal. The fault model of the system may state that the error routine will be invoked very infrequently – but when it is, it is urgent and hence it has a short deadline. Our model must therefore distinguish between D and T , and allow $D < T$. Indeed, for many periodic tasks, it is also useful to allow the application to define deadline values less than period.

An inspection of the response time algorithm for FPS, described in Section 11.5, reveals that:

- it works perfectly for values of D less than T as long as the stopping criterion becomes $w_i^{n+1} > D_i$;
- it works perfectly well with any priority ordering – $hp(i)$ always gives the set of higher-priority tasks.

Although some priority orderings are better than others, the test will provide the worst-case response times for the given priority ordering.

In Section 11.7, an optimal priority ordering for $D < T$ is defined (and proved). A later section will consider an extended algorithm and optimal priority ordering for the general case of $D < T$, $D = T$ or $D > T$.

11.6.1 Hard and soft tasks

For sporadic tasks, average and maximum arrival rates may be defined. Unfortunately, in many situations the worst-case figure is considerably higher than the average. Interrupts often arrive in bursts and an abnormal sensor reading may lead to significant additional computation. It follows that measuring schedulability with worst-case figures may lead to very low processor utilizations being observed in the actual running system. As a guideline for the minimum requirement, the following two rules should always be complied with.

- Rule 1 – all tasks should be schedulable using average execution times and average arrival rates.
- Rule 2 – all hard real-time tasks should be schedulable using worst-case execution times and worst-case arrival rates of all tasks (including soft).

A consequence of Rule 1 is that there may be situations in which it is not possible to meet all current deadlines. This condition is known as a **transient overload**; Rule 2, however, ensures that no hard real-time task will miss its deadline. If Rule 2 gives rise to unacceptably low utilizations for ‘normal execution’, direct action should be taken to try and reduce the worst-case execution times (or arrival rates).

11.6.2 Aperiodic tasks and fixed-priority execution-time servers

One simple way of scheduling aperiodic tasks, within a priority-based scheme, is to run such tasks at a priority below the priorities assigned to hard tasks. In effect, the aperiodic tasks run as background activities, and therefore cannot steal, in a preemptive system, resources from the hard tasks. Although a safe scheme, this does not provide adequate support to soft tasks which will often miss their deadlines if they only run as background activities. To improve the situation for soft tasks, a **server** (or **execution-time server**) can be employed. Servers protect the tasking resources needed by hard tasks, but otherwise allow soft tasks to run as soon as possible.

Since they were first introduced in 1987, a number of server methods have been defined. Here only two will be considered: the **Deferrable Server (DS)** and the **Sporadic Server (SS)** (Lehoczky et al., 1987).

With the DS, an analysis is undertaken (using, for example, the response time approach) that enables a new task to be introduced at the highest priority.³ This task, the

³ Servers at other priorities are possible, but the description is more straightforward if the server is given a higher priority than all the hard tasks.

server, thus has a period, T_s and a capacity C_s . These values are chosen so that all the hard tasks in the system remain schedulable even if the server executes periodically with period T_s and execution time C_s . At run-time, whenever an aperiodic task arrives, and there is capacity available, it starts executing immediately and continues until either it finishes or the capacity is exhausted. In the latter case, the aperiodic task is suspended (or transferred to a background priority). With the DS model, the capacity is replenished every T_s time units.

The operation of the SS differs from DS in its replenishment policy. With SS, if a task arrives at time t and uses c capacity then the server has this c capacity replenished T_s time units after t . In general, SS can furnish higher capacity than DS but has increased implementation overheads. Section 12.6 describes how SS is supported by C/Real-Time POSIX; DS and SS can be analysed using response time analysis (Bernat and Burns, 1999).

As all servers limit the capacity that is available to aperiodic soft tasks, they can also be used to ensure that sporadic tasks do not execute more often than expected. If a sporadic task with inter-arrival interval of T_i and worst-case execution time of C_i is implemented not directly as a task, but via a server with $T_s = T_i$ and $C_s = C_i$, then its impact (interference) on lower-priority tasks is bounded even if the sporadic task arrives too quickly (which would be an error condition).

All servers (DS, SS and others) can be described as *bandwidth preserving* in that they attempt to:

- make CPU resources available immediately to aperiodic tasks (if there is a capacity);
- retain the capacity for as long as possible if there are currently no aperiodic tasks (by allowing the hard tasks to execute).

Another bandwidth preserving scheme, which often performs better than the server techniques, is **dual-priority scheduling** (Davis and Wellings, 1995). Here, the range of priorities is split into three bands: high, medium and low. All aperiodic tasks run in the middle band. Hard tasks, when they are released, run in the low band, but they are promoted to the top band in time to meet their deadlines. Hence in the first stage of execution they will give way to aperiodic activities (but will execute if there is no such activity). In the second phase they will move to a higher priority and then have precedence over the aperiodic work. In the high band, priorities are assigned according to the deadline monotonic approach (see below). Promotion to this band occurs at time $D - R$. To implement the dual-priority scheme requires a dynamic priority provision.

11.7 Task systems with $D < T$

In the above discussion on sporadic tasks it was argued that, in general, it must be possible for a task to define a deadline that is less than its inter-arrival interval (or period). It was also noted earlier that for $D = T$ the rate monotonic priority ordering was optimal for a fixed priority scheme. Leung and Whitehead (1982) showed that

Task	Period, T	Deadline, D	Computation time, C	Priority, P	Response time, R
a	20	5	3	4	3
b	15	7	3	3	6
c	10	10	4	2	10
d	20	20	3	1	20

Table 11.10 Example task set for DMPO.

for $D < T$, a similar formulation could be defined – the **deadline monotonic** priority ordering (DMPO). Here, the fixed priority of a task is inversely proportional to its relative deadline: $(D_i < D_j \Rightarrow P_i > P_j)$. Table 11.10 gives the appropriate priority assignments for a simple task set. It also includes the worst-case response time – as calculated by the algorithm in Section 11.5. Note that a rate monotonic priority ordering would successfully schedule these tasks.

In the following subsection, the optimality of DMPO is proven. Given this result and the direct applicability of response time analysis to this task model, it is clear that FPS can adequately deal with this more general set of scheduling requirements. The same is not true for EDF scheduling, see Section 11.11. Once tasks can have $D < T$ then the simple utilization test (total utilization less than one) cannot be applied.

Having raised this difficulty with EDF, it must be remembered that EDF is the more effective scheduling scheme. Hence any task set that passes an FPS schedulability test *will* also always meet its timing requirements if executed under EDF. The necessary and sufficient tests for FPS can thus be seen as sufficient tests for EDF.

11.7.1 Proof that DMPO is optimal

Deadline monotonic priority ordering is optimal if any task set, Q , that is schedulable by priority scheme, W , is also schedulable by DMPO. The proof of optimality of DMPO will involve transforming the priorities of Q (as assigned by W) until the ordering is DMPO. Each step of the transformation will preserve schedulability.

Let i and j be two tasks (with adjacent priorities) in Q such that under W : $P_i > P_j$ and $D_i > D_j$. Define scheme W' to be identical to W except that tasks i and j are swapped. Consider the schedulability of Q under W' .

- All tasks with priorities greater than P_i will be unaffected by this change to lower-priority tasks.
- All tasks with priorities lower than P_j will be unaffected. They will all experience the same interference from i and j .
- Task j , which was schedulable under W , now has a higher priority, suffers less interference, and hence must be schedulable under W' .

All that is left is the need to show that task i , which has had its priority lowered, is still schedulable.

Under W , $R_j \leq D_j$, $D_j < D_i$ and $D_i \leq T_i$ and hence task i only interferes once during the execution of j .

Once the tasks have been switched, the new response time of i becomes equal to the old response time of j . This is true because under both priority orderings $C_j + C_i$ amount of computation time has been completed with the same level of interference from higher-priority tasks. Task j was released only once during R_j , and hence interferes only once during the execution of i under W' . It follows that:

$$R'_i = R_j \leq D_j < D_i$$

It can be concluded that task i is schedulable after the switch.

Priority scheme W' can now be transformed (to W'') by choosing two more tasks 'that are in the wrong order for DMPO' and switching them. Each such switch preserves schedulability. Eventually there will be no more tasks to switch; the ordering will be exactly that required by DMPO and the task set will still be schedulable. Hence, DMPO is optimal.

Note that for the special case of $D = T$, the above proof can be used to show that, in this circumstance, rate monotonic ordering is also optimal.

11.8 Task interactions and blocking

One of the simplistic assumptions embodied in the system model, described in Section 11.2.4, is the need for tasks to be independent. This is clearly unreasonable, as task interaction will be needed in almost all meaningful applications. In Chapters 5 and 6, it was noted that tasks can interact safely either by some form of protected shared data (using, for example, semaphores, monitors, synchronized methods or protected objects) or directly (using some form of rendezvous). All of these language features lead to the possibility of a task being suspended until some necessary future event has occurred (for example, waiting to gain a lock on a semaphore, or entry to a monitor, or until some other task is in a position to accept a rendezvous request). In general, synchronous communication leads to more pessimistic analysis as it is harder to define the real worst case when there are many dependencies between task executions. The following analysis is therefore more accurate when related to asynchronous communication where tasks exchange data via protected shared resources. The majority of the material in the next two sections is concerned with fixed-priority scheduling. The issue of task interactions and EDF scheduling will be considered in Section 11.11.4.

If a task is suspended waiting for a lower-priority task to complete some required computation then the priority model is, in some sense, being undermined. In an ideal world, such **priority inversion** (Lauer and Satterwaite, 1979) (that is, a high-priority task having to wait for a lower-priority task) should not exist. However, it cannot, in general, be totally eliminated. Nevertheless, its adverse effects can be minimized. If a task is waiting for a lower-priority task, it is said to be **blocked**. In order to test for schedulability, blocking must be bounded and measurable; it should also be small.

To illustrate an extreme example of priority inversion, consider the executions of four periodic tasks: a , b , c and d . Assume they have been assigned priorities according to the deadline monotonic scheme, so that the priority of task d is the highest and that of task a the lowest. Further, assume that tasks d and a (and tasks d and c) share a critical

Task	Priority	Execution sequence	Release time
<i>a</i>	1	<i>EQQQQE</i>	0
<i>b</i>	2	<i>EE</i>	2
<i>c</i>	3	<i>EVVE</i>	2
<i>d</i>	4	<i>EEQVE</i>	4

Table 11.11 Execution sequences.

section (resource), denoted by the symbol *Q* (and *V*), protected by mutual exclusion. Table 11.11 gives the details of the four tasks and their execution sequences; in this table ‘*E*’ represents a single tick of execution time and ‘*Q*’ (or ‘*V*’) represent an execution tick with access to the *Q* (or *V*) critical section. Thus task *c* executes for four ticks; the middle two while it has access to critical section *V*.

Figure 11.6 illustrates the execution sequence for the start times given in the table. Task *a* is released first, executes and locks the critical section, *Q*. It is then preempted by the release of task *c* which executes for one tick, locks *V* and is then preempted by the release of task *d*. The higher-priority task then executes until it also wishes to lock the critical section, *Q*; it must then be suspended (as the section is already locked by *a*). At this point, *c* will regain the processor and continue. Once it has terminated, *b* will commence and run for its entitlement. Only when *b* has completed will *a* be able to execute again; it will then complete its use of the *Q* and allow *d* to continue and complete. With this behaviour, *d* finishes at time 16, and therefore has a response time of 12; *c* has a value of 6, *b* a value of 8, and *a* a value of 17.

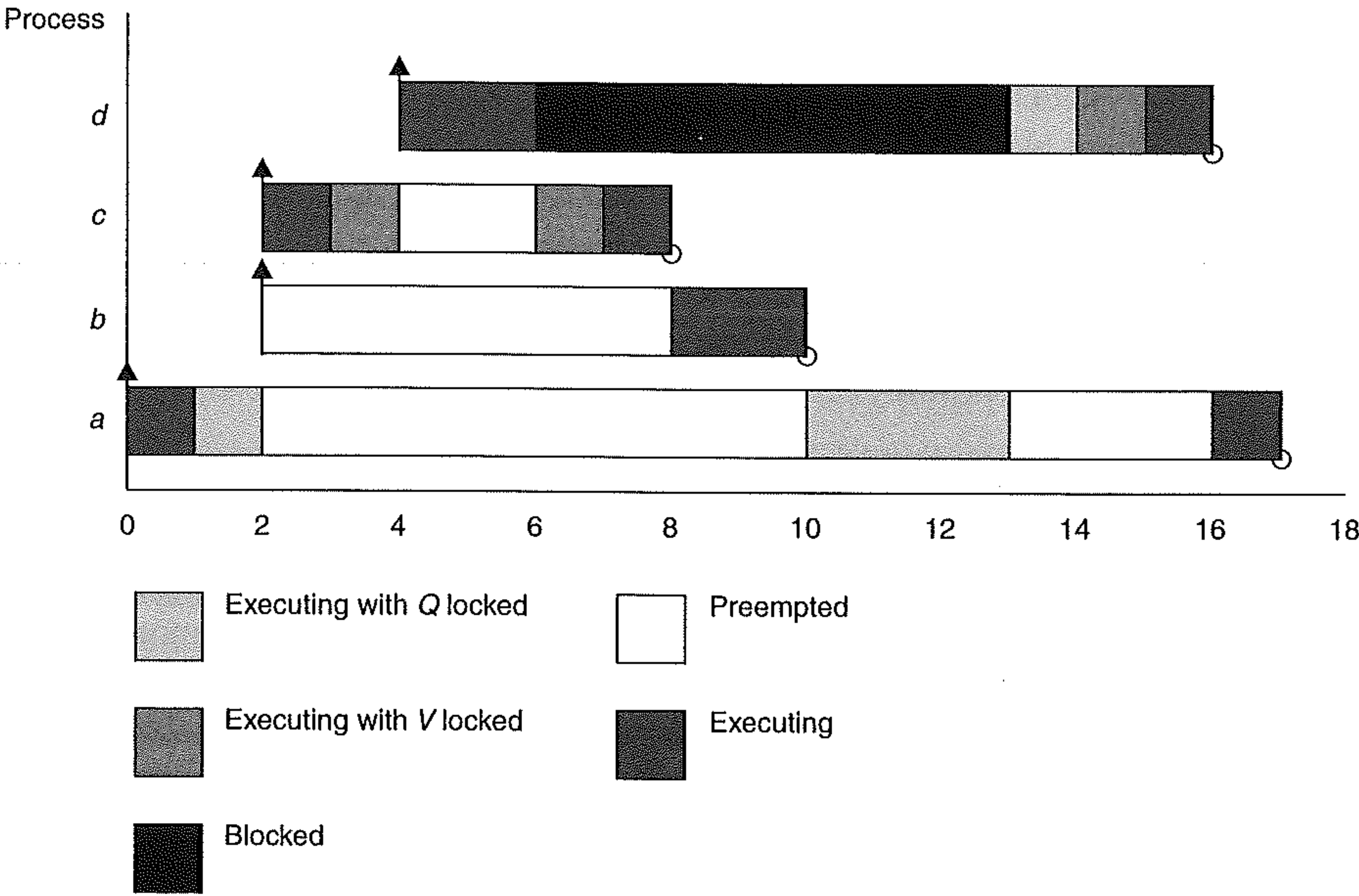


Figure 11.6 Example of priority inversion.

An inspection of Figure 11.6 shows that task d suffers considerable priority inversion. Not only is it blocked by task a but also by tasks b and c . Some blocking is inevitable; if the integrity of the critical section (and hence the shared data) is to be maintained then a must run in preference to d (while it has the lock). But the blocking of d by tasks c and b is unproductive and will severely affect the schedulability of the system (as the blocking on task d is excessive).

Priority inversion is not just a theoretical problem; real systems have been known to fail due to this phenomenon. A much publicized⁴ case was that of the NASA Mars Pathfinder. Although the Sojourner rover successfully survived the bouncy landing on Mars and was able to collect meteorological data, the spacecraft initially experienced a series of total system resets resulting in lost data. Tasks on the Pathfinder spacecraft were executed as fixed-priority threads. The high-priority data bus management thread and a low-priority meteorological data gathering thread shared an 'information bus' protected by a mutex. A communications thread ran with medium priority. At run-time, the release pattern of the threads was such that the high-priority thread was waiting for the mutex to be released on the information bus, but the lower-priority thread which was using the bus and hence held the mutex lock could not make progress as it was preempted by the relatively long-running medium-priority thread. This resulted in a watchdog timer being triggered as the urgent high-priority data bus thread was missing its deadline. The watchdog initiated a total system reset. The situation then repeated itself again and again. The solution to this problem, once it was identified (which was not easy), was to turn on **priority inheritance** that was fortunately supported by the spacecraft's operating system.

With priority inheritance, a task's priority is no longer static; if a task p is suspended waiting for task q to undertake some computation then the priority of q becomes equal to the priority of p (if it was lower to start with). In the example given a little earlier, task a will be given the priority of task d and will, therefore, run in preference to task c and task b . This is illustrated in Figure 11.7. Note that as a consequence of this algorithm, task b will now suffer blocking even though it does not use a shared object. Also note that task d now has a second block, but its response time has been reduced to 9. With the Mars Pathfinder example once priority inheritance was turned on, the lower-priority thread inherited the data bus thread's priority and thus ran in preference to the medium-priority thread.

With this simple inheritance rule, the priority of a task is the maximum of its own default priority and the priorities of all the other tasks that are at that time dependent upon it.

In general, inheritance of priority is not restricted to a single step. If task d is waiting for task c , but c cannot deal with d because it is waiting for task b then b as well as c is given d 's priority.

In the design of a real-time language, priority inheritance is of paramount importance. To have the most effective model, however, implies that the concurrency model should have a particular form. With standard semaphores and condition variables, there is no direct link between the act of becoming suspended and the identity of the task that will reverse this action. Inheritance is therefore not easily implemented. With synchronous

⁴See http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html.

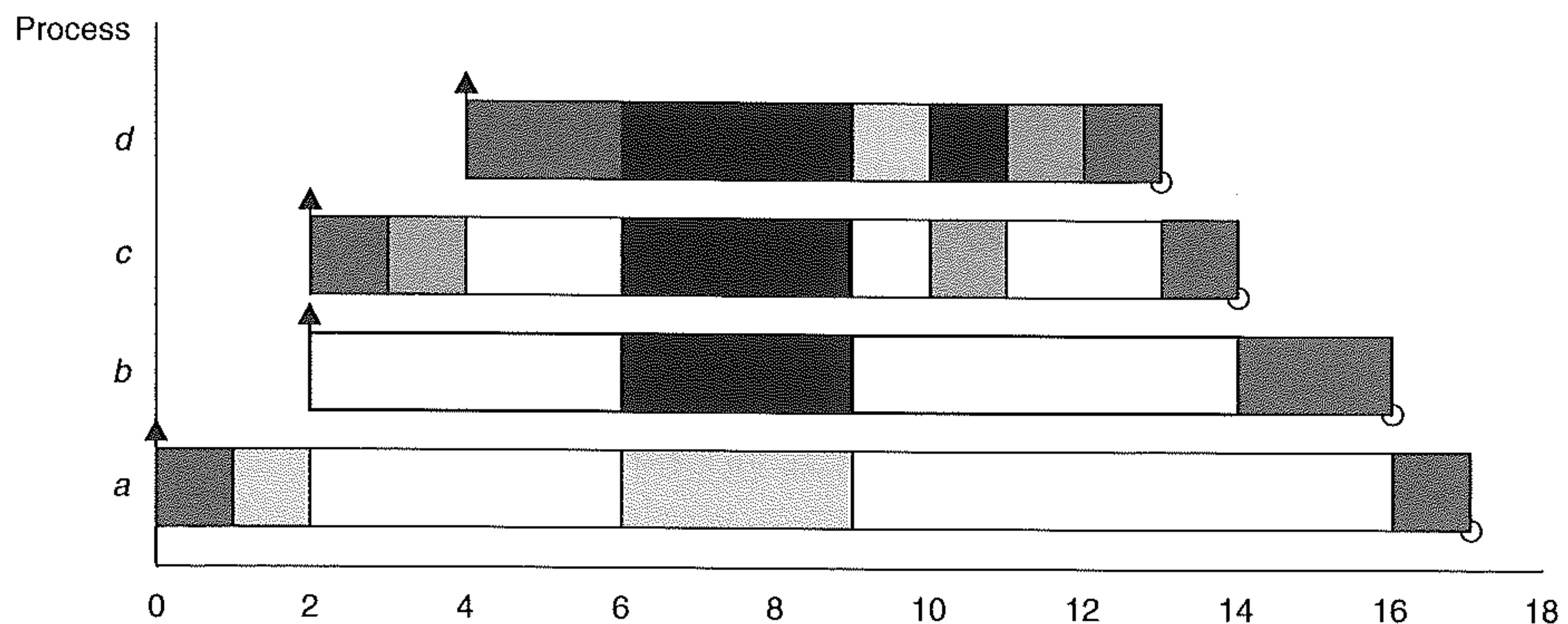


Figure 11.7 Example of priority inheritance.

message passing, indirect naming may also make it difficult to identify the task upon which one is waiting. To maximize the effectiveness of inheritance, direct symmetric naming would be the most appropriate.

Sha et al. (1990) show that with a priority inheritance protocol, there is a bound on the number of times a task can be blocked by lower-priority tasks. If a task has m critical sections that can lead to it being blocked then the maximum number of times it can be blocked is m . That is, in the worst case, each critical section will be locked by a lower-priority task (this is what happened in Figure 11.7). If there are only n ($n < m$) lower-priority tasks then this maximum can be further reduced (to n).

If B_i is the maximum blocking time that task i can suffer then for this simple priority inheritance model, a formula for calculating B can easily be found. Let K be the number of critical sections (resources) in the system. Equation (11.6) thus provides an upper bound on B :

$$B_i = \sum_{k=1}^K usage(k, i)C(k) \quad (11.6)$$

where $usage$ is a 0/1 function: $usage(k, i) = 1$ if resource k is used by at least one task with a priority less than P_i , and at least one task with a priority greater or equal to P_i . Otherwise it gives the result 0. $C(k)$ is the worst-case execution time of the k critical section. Nested resources are not accommodated by this simple formula; they require the $usage$ function to track resources that use other resources.

This algorithm is not optimal for this simple inheritance protocol. Firstly, it assumes a single cost for using the resource, it does not try to differentiate between the cost of each task's use of the resource. Secondly, it adds up the blocking from each resource, but this can only happen if each such resource is used by a different lower-priority process. This may not be possible for a particular application. For example, if all k resources are only used by one lower-priority task then there would be just one term to include in the equation for B . Nevertheless, the equation serves to illustrate the factors that need to be taken into account when calculating B . In Section 11.9, better inheritance protocols will be described and an improved formula for B will be given.

11.8.1 Response time calculations and blocking

Given that a value for B has been obtained, the response time algorithm can be modified to take the blocking factor into account:⁵

$$R = C + B + I$$

that is,

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad (11.7)$$

which can again be solved by constructing a recurrence relationship:

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \quad (11.8)$$

Note that this formulation may now be pessimistic (that is, not necessarily sufficient and necessary). Whether a task actually suffers its maximum blocking will depend upon task phasings. For example, if all tasks are periodic and all have the same period then no preemption will take place and hence no priority inversion will occur. However, in general, Equation (11.7) represents an effective scheduling test for real-time systems containing cooperating tasks.

11.9 Priority ceiling protocols

While the standard inheritance protocol gives an upper bound on the number of blocks a high-priority task can encounter, this bound can still lead to an unacceptably pessimistic worst-case calculation. This is compounded by the possibility of chains of blocks developing (transitive blocking), that is, task c being blocked by task b which is blocked by task a and so on. As shared data is a system resource, from a resource management point of view not only should blocking be minimized, but failure conditions such as deadlock should be eliminated. All of these issues are addressed by the ceiling priority protocols (Sha et al., 1990), two of which will be considered in this chapter: the **original ceiling priority protocol** and the **immediate ceiling priority protocol**. The original protocol (OCP) will be described first, followed by the somewhat more straightforward immediate variant (ICPP). When either of these protocols is used on a single-processor system:

- a high-priority task can be blocked at most once during its execution by lower-priority tasks;
- deadlocks are prevented;
- transitive blocking is prevented;
- mutual exclusive access to resources is ensured (by the protocol itself).

⁵Blocking can also be incorporated into the utilization-based tests, but now each task must be considered individually.

The ceiling protocols can best be described in terms of resources protected by critical sections. In essence, the protocol ensures that if a resource is locked, by task a say, and could lead to the blocking of a higher-priority task (b), then no other resource that could block b is allowed to be locked by any task other than a . A task can therefore be delayed by not only attempting to lock a previously locked resource but also when the lock could lead to multiple blocking on higher-priority tasks.

The original protocol takes the following form.

- (1) Each task has a static default priority assigned (perhaps by the deadline monotonic scheme).
- (2) Each resource has a static ceiling value defined; this is the maximum priority of the tasks that use it.
- (3) A task has a dynamic priority that is the maximum of its own static priority and any it inherits due to it blocking higher-priority tasks.
- (4) A task can only lock a resource if its dynamic priority is higher than the ceiling of any currently locked resource (excluding any that it has already locked itself).

The locking of a first system resource is allowed. The effect of the protocol is to ensure that a second resource can only be locked if there does not exist a higher-priority task that uses both resources. Consequently, the maximum amount of time a task can be blocked is equal to the execution time of the longest critical section in any of the lower-priority tasks that are accessed by higher-priority tasks; that is, Equation (11.6) becomes:

$$B_i = \max_{k=1}^K usage(k, i)C(k) \quad (11.9)$$

The benefit of the ceiling protocol is that a high-priority task can only be blocked once (per activation) by any lower-priority task. The penalty of this result is that more tasks will experience this block.

Not all the features of the algorithm can be illustrated by a single example, but the execution sequence shown in Figure 11.8 does give a good indication of how the algorithm works and provides a comparison with the earlier approaches (that is, this figure illustrates the same task sequence used in Figures 11.6 and 11.7).

In Figure 11.8, task a again locks the first critical section, as no other resources have been locked. It is again preempted by task c , but now the attempt by c to lock the second section (V) is not successful as its priority (3) is not higher than the current ceiling (which is 4, as Q is locked and is used by task d). At time 3, a is blocking c , and hence runs with its priority at the level 3, thereby blocking b . The higher-priority task, d , preempts a at time 4, but is subsequently blocked when it attempts to access Q . Hence a will now continue (with priority 4) until it releases its lock on Q and has its priority drop back to 1. Now, d can continue until it completes (with a response time of 7).

The priority ceiling protocols ensure that a task is only blocked once during each invocation. Figure 11.8, however, appears to show task b (and task c) suffering two blocks. What is actually happening is that a single block is being broken in two by the preemption of task d . Equation (11.9) determines that all tasks (apart from task a) will suffer a maximum single block of 4. Figure 11.8 shows that for this particular execution sequence task c and task b actually suffer a block of 3 and task d a block of only 2.

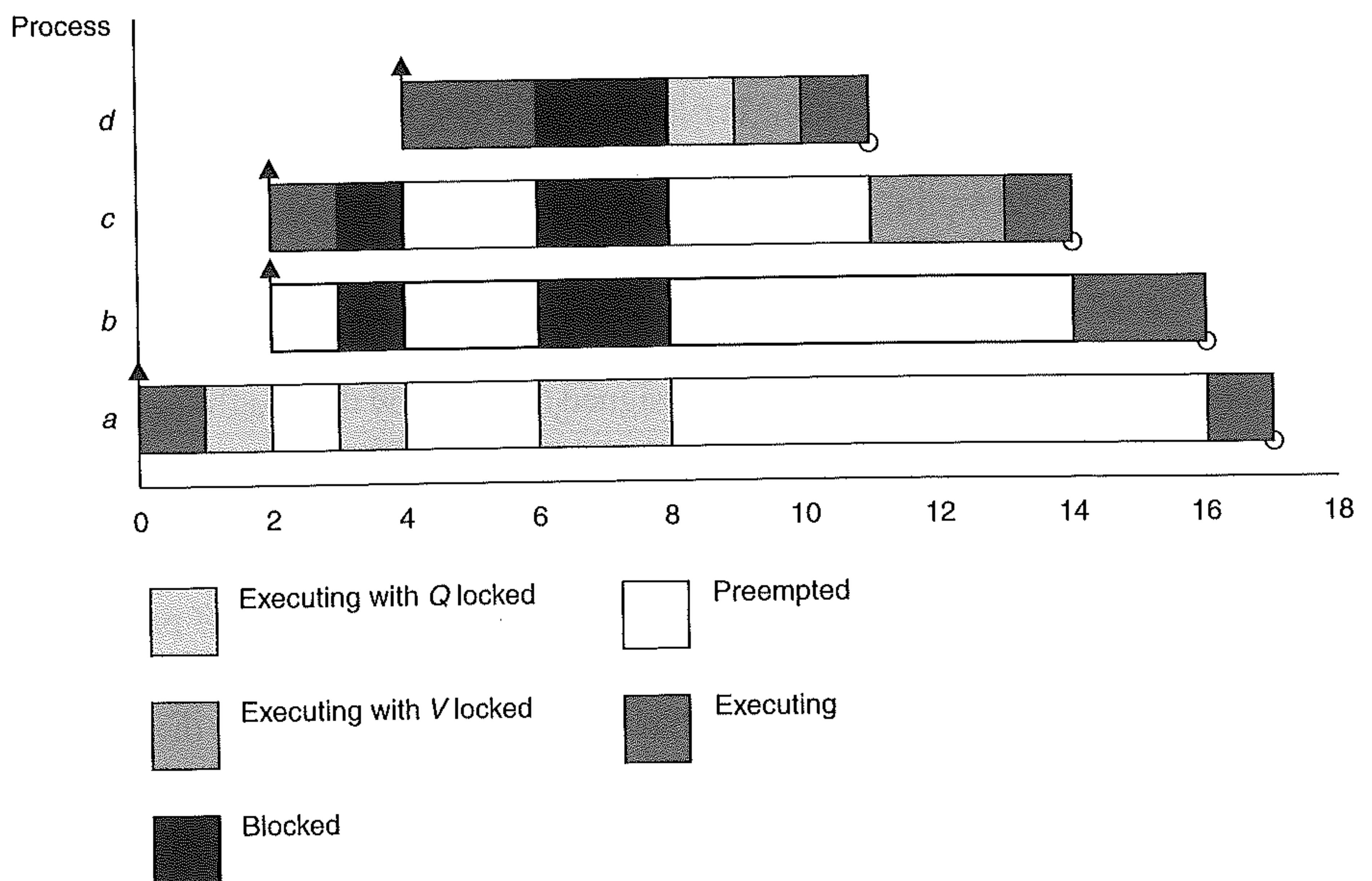


Figure 11.8 Example of priority inheritance – OCPP.

11.9.1 Immediate ceiling priority protocol

The immediate ceiling priority algorithm (ICPP) takes a more straightforward approach and raises the priority of a task as soon as it locks a resource (rather than only when it is actually blocking a higher-priority task). The protocol is thus defined as follows.

- Each task has a static default priority assigned (perhaps by the deadline monotonic scheme).
- Each resource has a static ceiling value defined; this is the maximum priority of the tasks that use it.
- A task has a dynamic priority that is the maximum of its own static priority and the ceiling values of any resources it has locked.

As a consequence of this final rule, a task will only suffer a block at the very beginning of its execution. Once the task starts actually executing, all the resources it needs must be free; if they were not, then some task would have an equal or higher priority and the task's execution would be postponed. The same task set used in earlier illustrations can now be executed under ICPP (see Figure 11.9).

Task *a* having locked *Q* at time 1, runs for the next four ticks with priority 4. Hence neither task *b*, task *c* nor task *d* can begin. Once *a* unlocks *Q* (and has its priority reduced), the other tasks execute in priority order. Note that all blocking is before actual execution and that *d*'s response time is now only 6. This is somewhat misleading, however, as the worst-case blocking time for the two protocols is the same (see Equation (11.9)).

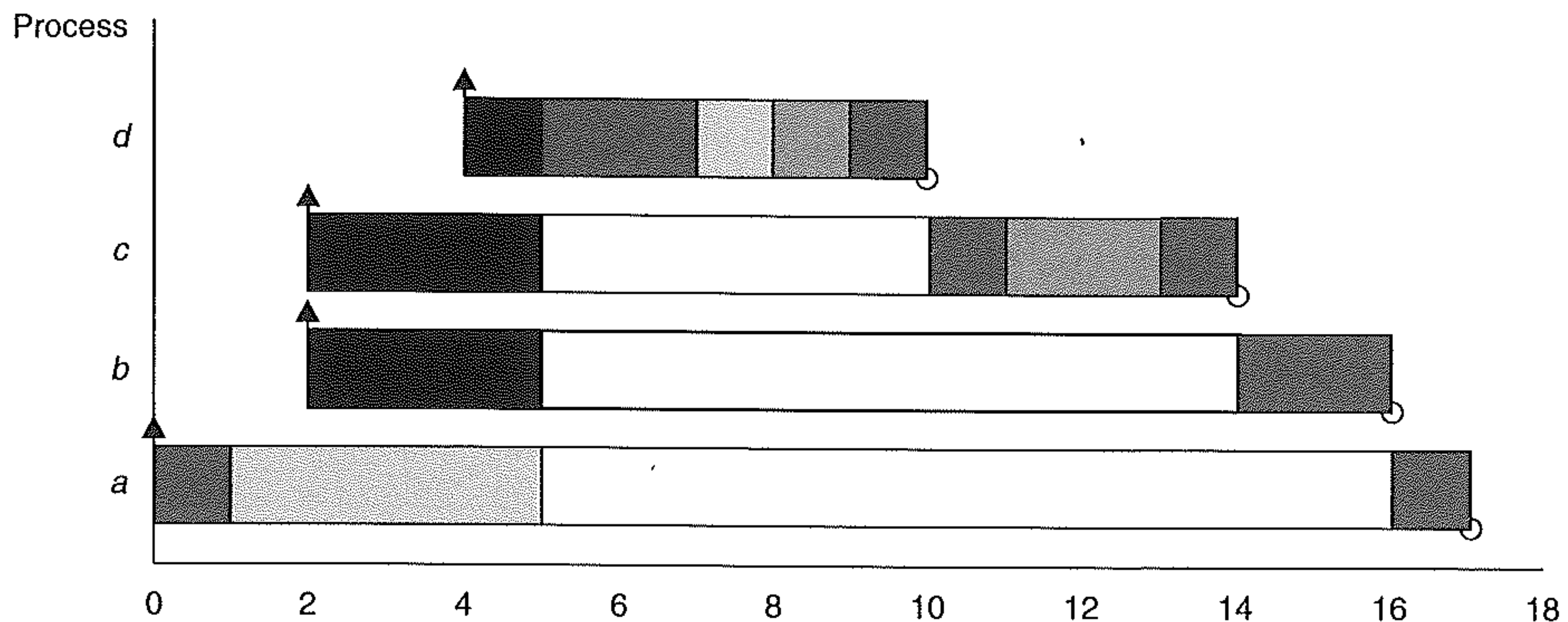


Figure 11.9 Example of priority inheritance – ICCP.

Although the worst-case behaviour of the two ceiling schemes is identical (from a scheduling view point), there are some points of difference.

- ICCP is easier to implement than the original (OCP) as blocking relationships need not be monitored.
- ICCP leads to fewer context switches as blocking is prior to first execution.
- ICCP requires more priority movements as this happens with all resource usages; OCP changes priority only if an actual block has occurred.

Finally, note that ICCP is called the Priority Protect protocol in C/Real-Time POSIX and Priority Ceiling Emulation in Real-Time Java.

11.9.2 Ceiling protocols, mutual exclusion and deadlock

Although the above algorithms for the two ceiling protocols were defined in terms of locks on resources, it must be emphasized that the protocols themselves rather than some other synchronization primitive provided the mutual exclusion access to the resource (at least on a single processor system and assuming the tasks do not suspend whilst holding a lock). Consider ICCP; if a task has access to some resource then it will be running with the ceiling value. No other task that uses that resource can have a higher priority, and hence the executing task will either execute unimpeded while using the resource, or, if it is preempted, the new task will not use this particular resource. Either way, mutual exclusion is ensured.

The other major property of the ceiling protocols (again for single-processor systems and non-self-suspension) is that they are deadlock-free. In Section 8.7, the issue of deadlock-free resource usage was considered. The ceiling protocols are a form of deadlock prevention. If a task holds one resource while claiming another, then the ceiling of the second resource cannot be lower than the ceiling of the first. Indeed, if two resources are used in different orders (by different tasks) then their ceilings must be identical. As one task is not preempted by another with merely the same priority, it follows that once

a task has gained access to a resource then all other resources will be free when needed. There is no possibility of circular waits and deadlock is prevented.

11.10 An extendible task model for FPS

It was noted earlier that the model outlined in Section 11.2.4 was too simplistic for practical use. In subsequent sections, three important restrictions were removed.

- Deadlines can be less than period ($D < T$).
- Sporadic and aperiodic tasks, as well as periodic tasks, can be supported.
- Task interactions are possible, with the resulting blocking being factored into the response time equations.

Within this section, five further generalizations will be given. The section will conclude with a general-purpose priority assignment algorithm.

11.10.1 Release jitter

In the simple model, all tasks are assumed to be periodic and to be released with perfect periodicity; that is, if task l has period T_l then it is released with exactly that frequency. Sporadic tasks are incorporated into the model by assuming that their minimum inter-arrival interval is T . This is not, however, always a realistic assumption. Consider a sporadic task s being released by a periodic task l (on another processor). The period of the first task is T_l and the sporadic task will have the same rate, but it is incorrect to assume that the maximum load (interference) s exerts on low-priority tasks can be represented in Equation (11.4) or (11.5) as a periodic task with period $T_s = T_l$.

To understand why this is insufficient, consider two consecutive executions of task l . Assume that the event that releases task s occurs at the very end of the periodic task's execution. On the first execution of task l , assume that the task does not complete until its latest possible time, that is, R_l . However, on the next invocation assume there is no interference on task l so it completes within C_l . As this value could be arbitrarily small, let it equal zero. The two executions of the sporadic task are not separated by T_l but by $T_l - R_l$. Figure 11.10 illustrates this behaviour for T_l equal to 20, R_l equal to 15 and minimum C_l equal to 1 (that is, two releases of the sporadic task within 6 time units). Note that this phenomenon is of interest only if task l is remote. If this was not the case then the variations in the release of task s would be accounted for by the standard equations, where a critical instant can be assumed between the releaser and the released.

To capture correctly the interference sporadic tasks have upon other tasks, the recurrence relationship must be modified. The maximum variation in a task's release is termed its **release jitter** (and is represented by J). For example, in the above, task s would have a jitter value of 15. In terms of its maximum impact on lower-priority tasks, this sporadic task will be released at time 0, 5, 25, 45 and so on. That is, at times 0, $T - J$, $2T - J$, $3T - J$, and so on. Examination of the derivation of the schedulability equation implies that task i will suffer one interference from task s if R_i is between 0 and $T - J$,

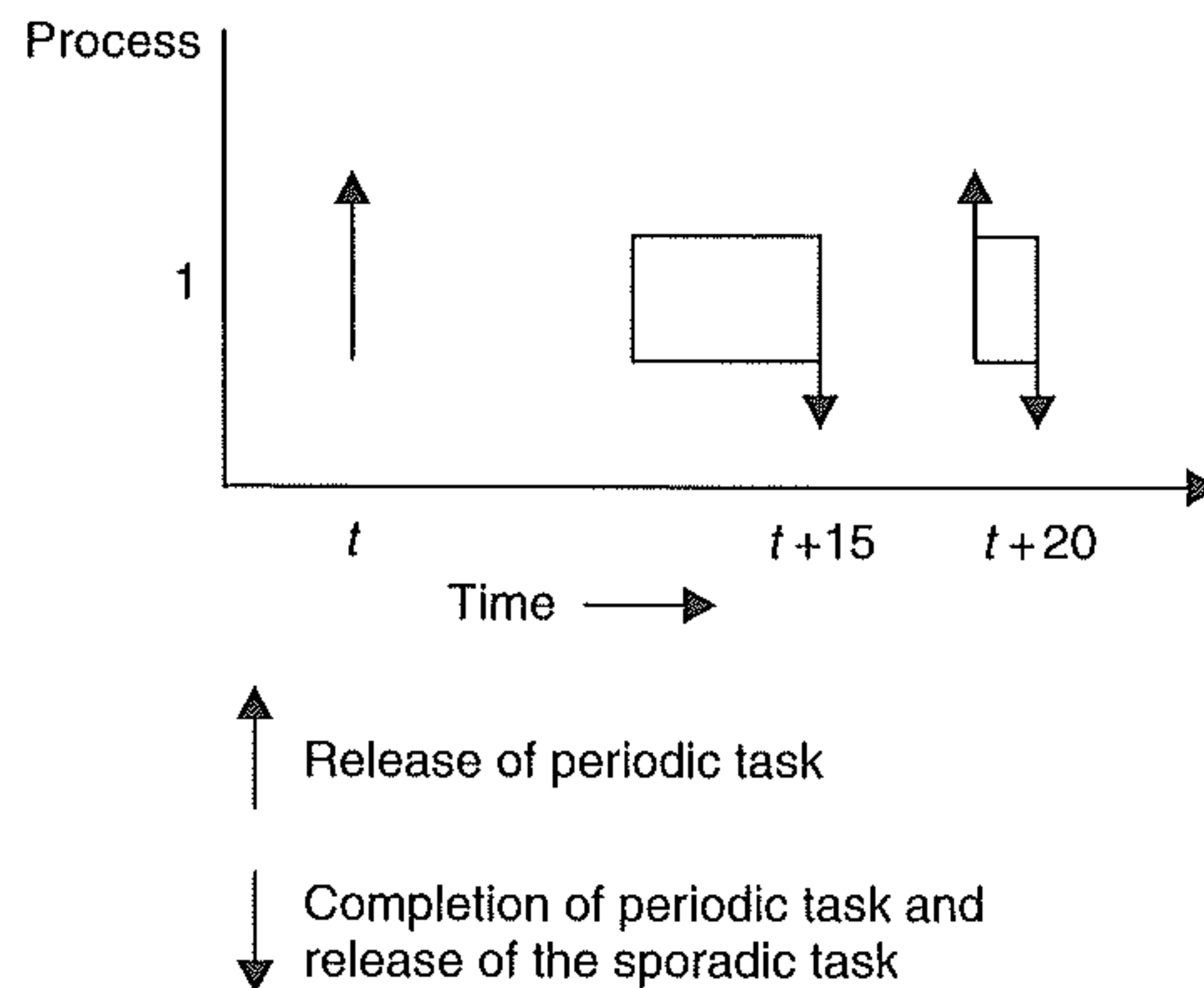


Figure 11.10 Releases of sporadic tasks.

that is $R_i \in [0, T - J)$, two if $R_i \in [T - J, 2T - J)$, three if $R_i \in [2T - J, 3T - J)$ and so on. A slight rearrangement of these conditions shows a single hit if $R_i + J \in [0, T)$, a double hit if $R_i + J \in [T, 2T)$ and so on. This can be represented in the same form as the previous response time equations as follows (Audsley et al., 1993b):

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \quad (11.10)$$

In general, periodic tasks do not suffer release jitter. An implementation may, however, restrict the granularity of the system timer (which releases periodic tasks). In this situation, a periodic task may also suffer release jitter. For example, a T value of 10 but a system granularity of 8 will imply a jitter value of 6 – at time 16 the periodic task will be released for its time ‘10’ invocation. If response time (now denoted as $R_i^{periodic}$) is to be measured relative to the real release time then the jitter value must be added to that previously calculated:

$$R_i^{periodic} = R_i + J_i \quad (11.11)$$

If this new value is greater than T_i then the following analysis must be used.

11.10.2 Arbitrary deadlines

To cater for situations where D_i (and hence potentially R_i) can be greater than T_i , the analysis must again be adapted. When deadline is less than (or equal) to period, it is necessary to consider only a single release of each task. The critical instant, when all higher-priority tasks are released at the same time, represents the maximum interference and hence the response time following a release at the critical instant must be the worst case. However, when deadline is greater than period, a number of releases must be considered. *The following assumes that the release of a task will be delayed until any previous releases of the same task have completed.*

If a task executes into the next period then both releases must be analysed to see which gives rise to the longest response time. Moreover, if the second release is not completed before a third occurs then this new release must also be considered, and so on.

For each potentially overlapping release, a separate window $w(q)$ is defined, where q is just an integer identifying a particular window (that is, $q = 0, 1, 2, \dots$). Equation (11.5) can be extended to have the following form (ignoring release jitter) (Tindell et al., 1994):

$$w_i^{n+1}(q) = B_i + (q + 1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j \quad (11.12)$$

For example, with q equal to 2, three releases of the task will occur in the window. For each value of q , a stable value of $w(q)$ can be found by iteration – as in Equation (11.5). The response time is then given as:

$$R_i(q) = w_i^n(q) - qT_i \quad (11.13)$$

For example, with $q = 2$ the task started $2T_i$ into the window and hence the response time is the size of the window minus $2T_i$.

The number of releases that need to be considered is bounded by the lowest value of q for which the following relation is true:

$$R_i(q) \leq T_i \quad (11.14)$$

At this point, the task completes before the next release and hence subsequent windows do not overlap. The worst-case response time is then the maximum value found for each q :

$$R_i = \max_{q=0,1,2,\dots} R_i(q) \quad (11.15)$$

Note that for $D \leq T$, the relation in Equation (11.14) is true for $q = 0$ (if the task can be guaranteed), in which case Equations (11.12) and (11.13) simplify back to the original equation. If any $R > D$, then the task is not schedulable.

When this arbitrary deadline formulation is combined with the effect of release jitter, two alterations to the above analysis must be made. First, as before, the interference factor must be increased if any higher-priority tasks suffer release jitter:

$$w_i^{n+1}(q) = B_i + (q + 1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q) + J_j}{T_j} \right\rceil C_j \quad (11.16)$$

The other change involves the task itself. If it can suffer release jitter then two consecutive windows could overlap if response time plus jitter is greater than period. To accommodate this, Equation (11.13) must be altered:

$$R_i(q) = w_i^n(q) - qT_i + J_i \quad (11.17)$$

11.10.3 Cooperative scheduling

The models described above have all required true preemptive dispatching. In this section, an alternative scheme is outlined (the use of deferred preemption). This has a number of advantages, but can still be analysed by the scheduling technique based on response time analysis. In Equation (11.7), for example, there is a blocking term B that accounts for the time a lower-priority task may be executing while a higher-priority task is runnable. In the application domain, this may be caused by the existence of data that is shared (under mutual exclusion) by tasks of different priority. Blocking can, however, also be caused by the run-time system or kernel. Many systems will have the non-preemptable context switch as the longest blocking time (for example, the release of a higher-priority task being delayed by the time it takes to context switch to a lower-priority task – even though an immediate context switch to the higher-priority task will then ensue).

One of the advantages of using the immediate ceiling priority protocol (to calculate and bound B) is that blocking is not cumulative. A task cannot be blocked both by an application task and a kernel routine – only one could actually be happening when the higher-priority task is released.

Cooperative scheduling exploits this non-cumulative property by increasing the situation in which blocking can occur. Let B_{MAX} be the maximum blocking time in the system (using a conventional approach). The application code is then split into non-preemptive blocks, the execution times of which are bounded by B_{MAX} . At the end of each of these blocks, the application code offers a ‘de-scheduling’ request to the kernel. If a high-priority task is now runnable the kernel will instigate a context switch; if not, the currently running task will continue into the next non-preemptive block.

The normal execution of the application code is thus totally cooperative. A task will continue to execute until it offers to de-schedule. Hence, as long as any critical section is fully contained between de-scheduling calls, mutual exclusion is assured. This method does, therefore, require the careful placement of de-scheduling calls.

To give some level of protection over corrupted (or incorrect) software, a kernel could use an asynchronous signal, or abort, to remove the application task if any non-preemptive block lasts longer than B_{MAX} (see Chapter 13).

The use of deferred preemption has two important advantages. It increases the schedulability of the system, and it can lead to lower values of C . In the solution of Equation (11.4), as the value of w is being extended, new releases of higher-priority tasks are possible that will further increase the value of w . With deferred preemption, no interference can occur during the last block of execution. Let F_i be the execution time of the final block, such that when the task has consumed $C_i - F_i$ time units, the last block has (just) started. Equation (11.4) is now solved for $C_i - F_i$ rather than C_i :

$$w_i^{n+1} = B_{\text{MAX}} + C_i - F_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \quad (11.18)$$

When this converges (that is, $w_i^{n+1} = w_i^n$), the response time is given by:

$$R_i = w_i^n + F_i \quad (11.19)$$

In effect, the last block of the task has executed with a higher priority (the highest) than the rest of the tasks.

This straightforward application of response time analysis is, however, misleading and may in certain circumstances lead to errors – that is, the analysis is not sufficient. Consider as a simple example a two task system with each task having deadline equal to period. The first task has a period of 6 and a computation time of 2 which is executed as a single non-preemptive block. The other task has a period of 8 and an execution time of 6 split into two 3 unit blocks. The longer period for this task means that it has the lower priority. The first task has a blocking term of 3 which, with its own computation time of 2, gives a response time of 5. The second task is first analysed to see when its first block will complete. This has a computation time of 3 and suffers 2 units of interference and so w_i^n converges simple to the value 5. To this is added the F_i value of 3 to give an overall response time of 8. This appears to imply that the system is schedulable. But this is impossible – the overall utilization of these two tasks is greater than 1 ($1/3 + 3/4$) which is indisputable evidence of unschedulability.

So why does the analysis fail on this example? There is a constraint on using Equations (11.18) and (11.19) that is hidden and this example highlights the problem because it does not satisfy this constraint. For these equations to apply, the worst-case response time for each task *with preemption* must be less than the task's period. If this is not the case then it is possible for the second (or third . . .) release of the task to be the worst. If releases overlap in this way then the analysis used in the previous section for deadline greater than period must be used.

For the example, the preemptive worst-case response time of the second task is 10 (two interferences plus execution time of 6) which is greater than 8 and hence the second release must be analysed. The easiest method for computing this is to look at the worst-case response time of a task made up of two serial executions of the second task. Now this new task has a computation time of 12 made up of four 3 unit blocks. Applying Equations (11.18) and (11.19) gives a value of w_i^n of 15; when the final 3 is added in this gives a response time of 18 which breaks the deadline value of 16 (for the second invocation).

It must be emphasized that for most systems with utilization not greater than 1, releases will not overlap and the straightforward use of these equations will provide the correct result – but the constraint must always be checked.

The other advantage of deferred preemption comes from predicting more accurately the execution times of a task's non-preemptable basic blocks. Modern processors have caches, prefetch queues and pipelines that all significantly reduce the execution times of code. Typically, simple estimations of worst-case execution time are forced to ignore these advantages and obtain very pessimistic results because preemption will invalidate caches and pipelines. Knowledge of non-preemption can be used to predict the speed up that will occur in practice. However, if the cost of postponing a context switch is high, this will militate against these advantages.

11.10.4 Fault tolerance

Fault tolerance via either forward or backward error recovery always results in extra computation. This could be an exception handler or a recovery block. In a real-time fault-tolerant system, deadlines should still be met even when a certain level of faults occur. This level of fault tolerance is known as the **fault model**. If C_i^f is the extra

computation time that results from an error in task i , then the response time equation can easily be changed:

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} C_k^f \tag{11.20}$$

where $hep(i)$ is the set of tasks with a priority equal to or higher than i .

Here, the fault model defines a maximum of one fault and there is an assumption that a task will execute its recovery action at the same priority as its ordinary computation. Equation (11.20) is easily changed to increase the number of allowed faults (F):

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} F C_k^f \tag{11.21}$$

Indeed, a system can be analysed for increasing values of F to see what number of faults (arriving in a burst) can be tolerated. Alternatively, the fault model may indicate a minimum arrival interval for faults. In this case the equation becomes:

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} \left(\left\lceil \frac{R_i}{T_f} \right\rceil C_k^f \right) \tag{11.22}$$

where T_f is the minimum inter-arrival time between faults.

In Equations (11.21) and (11.22), the assumption is made that in the worst case, the fault will always occur in the task that has the longest recovery time.

11.10.5 Introducing offsets

In the scheduling analysis presented so far in this chapter, it has been assumed that all tasks share a common release time. This critical instant is when all tasks are released simultaneously (this is usually taken to occur at time 0). For fixed-priority scheduling, this is a safe assumption; if all tasks meet their timing requirements when released together then they will always be schedulable. There are, however, sets of periodic tasks that can benefit from explicitly choosing their release times so that they do not share a critical instant. This may result in improved schedulability. One task is said to have an **offset** with respect to the others. Consider for illustration the three tasks defined in Table 11.12.

If a critical instant is assumed then task a has response time of 4 and task b has a response time of 8, but the third task has a worst-case response time of 16, which is

Task	T	D	C
a	8	5	4
b	20	10	4
c	20	12	4

Table 11.12 Example of a task set.

Task	<i>T</i>	<i>D</i>	<i>C</i>	<i>O</i>	<i>R</i>
<i>a</i>	8	5	4	0	4
<i>b</i>	20	10	4	0	8
<i>c</i>	20	12	4	10	8

Table 11.13 Response time analysis of the task set.

Task	<i>T</i>	<i>D</i>	<i>C</i>	<i>O</i>	<i>R</i>
<i>a</i>	8	5	4	0	4
<i>n</i>	10	10	4	0	8

Table 11.14 Response time analysis of the transformed task set.

beyond its deadline. For task *c* the interference from task *b* is sufficient to force a further interference from *a*, and this is crucial. However, if task *c* is given an offset (*O*) of 10 (that is, retain the same period and relative deadline, but have its first release at time 10) then it will never execute at the same time as *b*. The result is a schedulable task set – see Table 11.13.

Unfortunately, task sets with arbitrary offsets are not amenable to analysis. It is a strongly NP-hard problem to choose offsets so that a task set is optimally schedulable. Indeed, it is far from trivial to even check if a set of tasks with offsets share a critical instant.⁶

Notwithstanding this theoretical result, there are task sets that can be analysed in a relatively straightforward (although not necessarily optimal) way. In most realistic systems, task periods are not arbitrary but are likely to be related to one another. As in the example just illustrated, two tasks have a common period. In these situations it is easy to give one an offset (of $T/2$) and to analyse the resulting system using a transformation technique that removes the offset – and hence critical instant analysis applies. In the example, tasks *b* and *c* (*c* having the offset of 10) are replaced by a single notional task with period 10, computation time 4, deadline 10 but no offset. This notional task has two important properties.

- If it is schedulable (when sharing a critical instant with all other tasks), the two real tasks will meet their deadlines when one is given the half period offset.
- If all lower-priority tasks are schedulable when suffering interference from the notional task (and all other high-priority tasks), they will remain schedulable when the notional task is replaced by the two real tasks (one with the offset).

These properties follow from the observation that the notional task always uses more (or equal) CPU time than the two real tasks. Table 11.14 shows the analysis that would apply to the transformed task set. The notional task is given the name ‘*n*’ in this table.

⁶One interesting result is that a task set with co-prime periods will always have a critical instant no matter what offsets are chosen (Audsley and Burns, 1998).

More generally the parameters of the notional task are calculated from the real tasks a and b as follows:

$$T_n = T_a/2 \quad (\text{or } T_b/2 \text{ as } T_a = T_b)$$

$$C_n = \text{Max}(C_a, C_b)$$

$$D_n = \text{Min}(D_a, D_b)$$

$$P_n = \text{Max}(P_a, P_b)$$

where P denotes priority.

Clearly, what is possible for two tasks is also applicable to three or more tasks. A fuller description of these techniques is given in Bate and Burns (1997). In summary, although arbitrary offsets are effectively impossible to analyse, the judicious use of offsets and the transformation technique can return the analysis problem to one of a simple task set that shares a critical instant. All the analysis given in earlier sections of this chapter, therefore, applies.

In Section 10.5 offsets are used to control input and output jitter. Typically the input and output activities involve much less computation time than the 'middle' task that implements whatever algorithms are necessary to convert the input value to an output setting. To analyse this program structure it is acceptable to ignore offsets. As noted earlier a system that is schedulable when offsets are ignored remains schedulable when they are added to the implementation scheme.

11.10.6 Other characteristics

In addition to the characteristics discussed in the last few sections (e.g. release jitter, non-preemption, fault tolerance, arbitrary deadlines and offsets) there are many other task attributes that have been analysed in the fixed-priority scheduling literature. For example, tasks with precedence, tasks that must meet N in M deadlines (e.g. 4 in 5) but not every deadline, and tasks that have a set of C values (not just a single maximum). It is not necessary, however, to cover all these topics (and more) in order to complete this treatment of RTA. The key property of RTA is that it is extendable and configurable. New characteristics can be easily accommodated into the theory.

11.10.7 Priority assignment

The formulation given for arbitrary deadlines has the property that no simple algorithm (such as rate or deadline monotonic) gives the optimal priority ordering. In this section, a theorem and algorithm for assigning priorities in arbitrary situations is given. The theorem considers the behaviour of the lowest priority task (Audsley et al., 1993b).

Theorem *If task p is assigned the lowest priority and is feasible, then, if a feasible priority ordering exists for the complete task set, an ordering exists with task p assigned the lowest priority.*

The proof of this theorem comes from considering the schedulability equations – for example, Equation (11.12). If a task has the lowest priority, it suffers interference from all

higher-priority tasks. This interference is not dependent upon the actual ordering of these higher priorities. Hence if any task is schedulable at the bottom value it can be assigned that place, and all that is required is to assign the other $N - 1$ priorities. Fortunately, the theorem can be reapplied to the reduced task set. Hence through successive reapplication, a complete priority ordering is obtained (if one exists).

The following code in Ada implements the priority assignment algorithm;⁷ Set is an array of tasks that is notionally ordered by priority; Set(N) being the highest priority, Set(1) being the lowest. The procedure Task_Test tests to see whether task K is feasible at that place in the array. The double loop works by first swapping tasks into the lowest position until a feasible result is found; this task is then fixed at that position. The next priority position is then considered. If at any time the inner loop fails to find a feasible task, the whole procedure is abandoned. Note that a concise algorithm is possible if an extra swap is undertaken.

```

procedure Assign_Pri (Set : in out Task_Set; N : Natural;
                      Ok : out Boolean) is
begin
  for K in 1..N loop
    for Next in K..N loop
      Swap(Set, K, Next);
      Task_Test(Set, K, Ok);
      exit when Ok;
    end loop;
    exit when not Ok;  -- failed to find a schedulable task
  end loop;
end Assign_Pri;

```

If the test of feasibility is exact (necessary and sufficient) then the priority ordering is optimal. Thus for arbitrary deadlines (without blocking), an optimal ordering is found. Where there is blocking, the priority ceiling protocols ensure that blockings are relatively small and, therefore, the above algorithm produces adequate near optimal results.

11.10.8 Insufficient priorities

In all of the analysis presented in this chapter it has been assumed that each task has a distinct priority. Unfortunately it is not always possible to accommodate this 'one priority per task' ideal. If there are insufficient priorities then two or more tasks must share the same priority. Fortunately, to check the schedulability of shared-priority tasks requires only a minor modification to the response time test. Consider the basis Equation (11.4) derived earlier in this chapter, which has a summation over all the higher-priority tasks. If tasks share priorities then this summation must be over all higher- or equal-priority tasks:

$$R_i = C_i + \sum_{j \in \text{hep}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad (11.23)$$

where $\text{hep}(i)$ is the set of higher- or equal-priority tasks (than i).

⁷This algorithm has become known as *Audsley's algorithm*.

So, if tasks a and b share priority i then a assumes it is getting interference from b and b assumes it is getting interference from a . Clearly if a and b are schedulable when they share priority i then they will remain schedulable if they are assigned distinct but adjacent priorities. The converse is, however, not true.

One way to reduce the number of priority levels required for a specific system is to first make sure the system is schedulable with distinct priorities. Then, starting from the lowest priority, tasks are grouped together until the addition of an extra task breaks schedulability. A new group is then started with this task, and the process continues until all tasks are in groups (although some groups may contain only have a single task). A minor variant of the priority assignment algorithm given above can easily implement this scheme.

Reducing the number of priority levels inevitably reduces schedulability. Tests have shown (Klein et al., 1993) that 92% of systems that are schedulable with distinct priorities will remain schedulable if only 32 levels are available. For 128 priority values this rises to 99%. It will be noted in the next chapter that Ada requires a minimum of 31 distinct priorities, Real-Time POSIX a minimum of 32 and Real-Time Java a minimum of 28.

11.10.9 Execution-time servers

Finally, in the description of fixed-priority scheduling the topic of execution-time servers is revisited. As applications and hardware platforms become more complicated it is useful to employ a virtual resource layer between the set of applications and the processor (or processors) they execute on. An execution-time server both guarantees a certain level of service and ensures that no more resource is allocated than is implied by the 'service contract'. So, for example, two multithreaded applications may co-exist on the one processor. One application receives 4 ms every 10 ms, the other 6 ms. These levels are guaranteed and policed. The first application will definitely get 4 ms, but it will not be allocated more than 4 ms in a 10 ms interval even if it has a runnable high-priority task.

There have been a number of execution-time servers proposed for FPS (see Section 11.6.2). Here three common ones are described: the **Periodic Server**, the **Deferrable Server** and the **Sporadic Server**. The simple Periodic Server has a budget (capacity) and a replenishment period. Following replenishment, client tasks can execute until either the budget is exhausted or there are no longer any runnable client tasks. The server is then suspended until the next replenishment time. The Deferrable Server is similar except that the budget remains available even after clients have been satisfied – a client arriving late will be serviced if there is budget available. Both Periodic and Deferrable Servers are replenished periodically and the budget still available at replenishment is lost. The Sporadic Server behaves a little differently. The budget remains indefinitely. When a client arrives (at time t , say) it can use up the available budget which is then replenished at time $t +$ the replenishment period of the server.

A complete system can contain a number of servers of these three types. The Periodic Server is ideally suited for supporting periodic tasks, the Sporadic Server is exactly what is required for sporadic tasks and the Deferrable Server is a good match for handling aperiodic work. In the latter case, aperiodic tasks can be handled quickly if

there is budget available – but once this is exhausted then the aperiodic tasks will not be serviced and hence an unbounded load on the server will have no detrimental effects on other parts of the system.

Scheduling these three server types on a fixed-priority system is relatively straightforward. Each server, of whichever type, is allocated a distinct priority. Response time analysis is then used to verify that all servers can guarantee their budget and replenishment period. Fortunately Periodic and Sporadic Servers behave exactly the same as periodic tasks and hence the straightforward analysis for these servers is directly applicable. For Deferrable Servers, the worst-case impact such a server can have on lower-priority tasks occurs when its budget is used at the very end of one period and then again at the start of the next. Conveniently this behaviour is identical to a periodic task suffering release jitter and hence can be analysed using the formulation given in Section 11.10.1.

It follows from this brief discussion that the schedulability test for a task running on a server involves two steps; first to verify that the server's parameters are valid and second that the response time of the task on that server is bounded by the task's deadline. The worst-case response time for a task executing on a server can be computed in a number of ways. For example, a server that guarantees 2 ms every 10 ms is equivalent to a processor running at 1/5 of its original speed. If all task computation times are multiplied by 5 then standard RTA can be applied (using these new C values).⁸ A similar approach is taken with variable speed processors – an example of this analysis is given in Section 11.15.

11.11 Earliest deadline first (EDF) scheduling

FPS is undoubtedly the most popular scheduling approach available to the implementors of real-time systems. The next chapter will show how it is supported in a number of languages and operating systems. However, as discussed at the beginning of this chapter, it is not the only approach studied in the real-time scheduling community. This section focuses on an alternative approach, EDF, that has a number of properties that make it almost as important as FPS. Unfortunately, it is currently less supported by languages and operating systems (again see the next chapter). For this reason EDF analysis is not covered here to the same level of detail afforded to FPS analysis.

11.11.1 Utilization-based schedulability tests for EDF

Not only did the seminal paper of Liu and Layland introduce a utilization-based test for FPS but it also gave one for EDF. The following equation is for the simple task model introduced in Section 11.2.4 – in particular, $D = T$ for all tasks:

$$\sum_{i=1}^N \left(\frac{C_i}{T_i} \right) \leq 1 \quad (11.24)$$

⁸With this example, the computed response times may need to have the value 8 added to take into account the 'dead time' before the sever can respond to requests from its client tasks.

Clearly this is a much simpler test than the corresponding test for FPS (Equation (11.1)). As long as the utilization of the task set is less than the total capacity of the processor then all deadlines will be met (for the simple task model). In this sense EDF is superior to FPS; it can always schedule any task set that FPS can, but not all task sets that are passed by the EDF test can be scheduled using fixed priorities. Given this advantage it is reasonable to ask why EDF is not the preferred task-based scheduling method? The reason is that FPS has a number of advantages over EDF.

- FPS is easier to implement, as the scheduling attribute (*priority*) is static; EDF is dynamic and hence requires a more complex run-time system which will have higher overhead.
- It is easier to incorporate tasks without deadlines into FPS (by merely assigning them a priority); giving a task an arbitrary deadline is more artificial.
- The deadline attribute is not the only parameter of importance; again it is easier to incorporate other factors into the notion of priority than it is into the notion of deadline, for example, the criticality of the task.
- During overload situations (which may be a fault condition) the behaviour of FPS is more predictable (the lower-priority tasks are those that will miss their deadlines first); EDF is unpredictable under overload and can experience a domino effect in which a large number of tasks miss deadlines. This is considered again in Section 11.12.
- The utilization-based test, for the simple model, is misleading as it is necessary and sufficient for EDF but only sufficient for FPS. Hence higher utilizations can, in general, be achieved for FPS.

Notwithstanding this final point, EDF does have an advantage over FPS because of its higher utilization. Indeed it is easy to show that if a task set, with restrictions such as deadline equal to period removed, is schedulable by any scheme then it will also be schedulable by EDF. The proof of this property follows the pattern used for proving that DMPO is optimal (see Section 11.7.1). Starting with the feasible schedule it is always possible to transform the schedule to one that becomes identical with the one EDF would produce – and at each transformation schedulability is preserved.

11.11.2 Processor demand criteria for EDF

One of the disadvantages of the EDF scheme is that the worst-case response time for each task does *not* occur when all tasks are released at a critical instant. In this situation only tasks with a shorter relative deadline will interfere. However, later there may exist a position in which all (or at least more) tasks have a shorter absolute deadline. In situations where the simple utilization-based test cannot be applied (for example when there is release jitter or when deadlines are shorter than periods) then a more sophisticated scheduling test must be used. In FPS this takes the form of RTA (calculate the worst-case response time for each task and then check that this is less than the related deadline). For EDF this approach can again be used, but it is much more complicated to calculate these response time values and hence it will not be described here. There is, however,

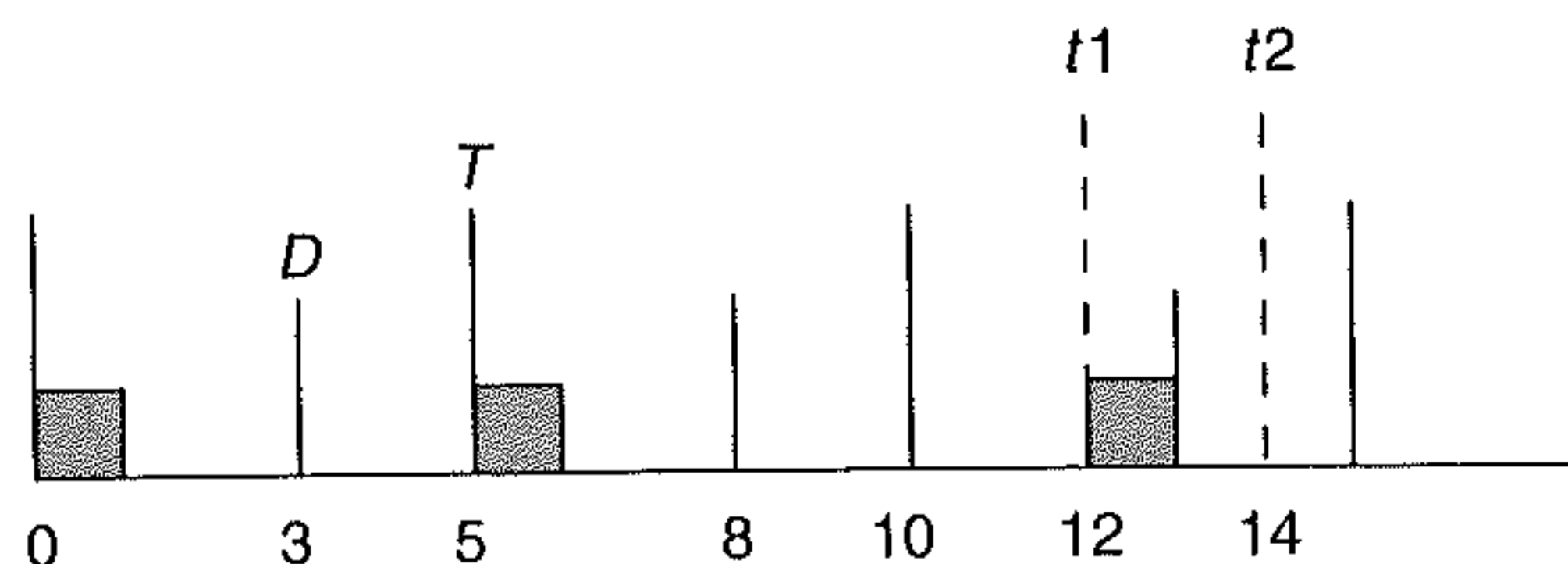


Figure 11.11 PDC example.

an alternative scheme that checks for schedulability directly rather than via response times. This method, called PDC (Processor Demand Criteria) (Baruah et al., 1990a, b), is defined as follows.

Assuming a system starts at time 0 and all tasks arrive at their maximum frequency, at any future time, t , it is possible to calculate the load on the system, $h(t)$. This is the amount of work that must be completed before t , in other words, all jobs that had absolute deadlines before (or at) t . It is easy to give a formula for $h(t)$:

$$h(t) = \sum_{i=1}^N \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor C_i \quad (11.25)$$

To illustrate this formula, consider a single task with $T = 5$, $D = 3$ and $C = 1$, and two possible values of t : $t1 = 12$ and $t2 = 14$. Figure 11.11 illustrates these parameters. The third deadline of the task is at 13 and hence is after $t1$ so $h(t1)$ for this task should be 2. But at $t2 = 14$ another execution of the task must be completed and so $h(t2) = 3$. To compute these values easily, $T - D$ is added to t ; if this results in a value after the next period then the floor function in Equation (11.25) will correctly add an extra C to the total. So, in the example, $T - D = 2$, $t1 + 2 = 14$ and hence $\lfloor 14/5 \rfloor = 2$. But $t2 + 2 = 16$ and so $\lfloor 16/5 \rfloor = 3$.

The requirement for schedulability is that the load must never exceed the time available to satisfy that level of load:

$$\forall t > 0 \quad h(t) \leq t \quad (11.26)$$

PDC involves applying this equation to a limited number of t values. The number of points is limited by two factors:

- only values of t that correspond to deadlines of tasks need be checked;
- there is an upper bound (L) on the values of t that must be checked – this means that an unschedulable system will have $h(t) > t$ for some value of $t < L$.

The first reduction comes from the fact that $h(t)$ is constant between deadlines and hence the worst case occurs at a deadline. To calculate the upper bound (L) on the interval that must be checked, two formulae have been developed. The first one comes from the need to check at least the first deadline of each task, and a bound based on utilization (the derivation of all the equations given in the section can be found in the

literature referenced in the Further Reading section at the end of this chapter):

$$L_a = \max \left\{ D_1, \dots, D_N, \frac{\sum_{i=1}^N (T_i - D_i) C_i / T_i}{1 - U} \right\}$$

The second bound is derived from the busy period of the system (that is, the time from start-up at time 0 to the first null or background tick where no task is executing – at this time, by definition, the load has been satisfied). This is easily obtained from a recurrence relation similar to that used in FPS:

$$w^0 = \sum_{i=1}^N C_i$$
$$w^{j+1} = \sum_{i=1}^N \left\lceil \frac{w^j}{T_i} \right\rceil C_i$$

When $w^{j+1} = w^j$ then this is the end of the busy period and $L_b = w^j$. Note this busy period is bounded if the utilization of the task set is not greater than 1 (so this is always checked first).

To obtain the least upper bound, the simple minimum of these two values is used:

$$L = \min(L_a, L_b) \tag{11.27}$$

For example, consider a three task system as depicted in Table 11.15. The utilization of this task set is 0.92. The computed values of L_a and L_b are 30.37 and 15 respectively; hence the least upper bound is 15. In the time period from 0 to 15 there are five deadlines to check – task a at times 4, 8 and 12, task b at time 10 and task c at time 14. At all of these points Equation (11.26) is satisfied and the system is determined to be schedulable. For example $h(15)$ is 14 and $h(12)$ is 6.

If the example is now modified to increase the computation time of b to 4 then the utilization is still acceptable (0.987), but it is not schedulable. At time 14, $h(t)$ has the value 15, so $h(14) > 14$.

11.11.3 The QPA test

For non-trivial systems, L can be large and the number of deadlines between 0 and L that need to be checked becomes excessive. Fortunately an efficient scheme has recently been developed that can significantly reduce the number of time points that need to be tested. This scheme, known as QPA (Quick Processor-demand Analysis), exploits the

Task	T	D	C
a	4	4	1
b	15	10	3
c	17	14	8

Table 11.15 A task set for EDF.

Task	<i>T</i>	<i>D</i>	<i>C</i>
<i>a</i>	4	4	1
<i>b</i>	8	6	1
<i>c</i>	10	10	1
<i>d</i>	12	8	2
<i>e</i>	15	12	2
<i>f</i>	21	20	3

Table 11.16 A task set for EDF.

following property (Zhang and Burns, 2008): rather than progress from 0 to L checking each deadline, QPA starts at L and moves backwards towards 0 checking only a necessary subset of the deadlines.

Let $h(L) = s$. If $s > L$ then the system is unschedulable. If this is not the case ($s \leq L$) then $h(t) < t$ for all values of t : $s < t < L$. Hence there is no need to check the deadlines within the interval $s..L$. To verify this property assume (in order to construct a counterexample) a value t within the range has $h(t) > t$. Now $t > s$ so $h(t) > s$. Also $h(t) < h(L)$ as $t < L$ (the function h is monotonic in t). We must conclude that $s > L$ which contradicts the assumption that $s < L$.

Having jumped from L back to $h(L)$, the procedure is repeated from $h(L)$ to $h(h(L))$, etc. At each step the essential test of $h(t) < t$ is undertaken. Of course if $h(t) = t$ then no progress can be made and it is necessary to force progress by moving from t to the largest absolute deadline (d) in the system such as $d < t$.

The QPA test looks at only a small fraction of the number of points that would need to be analysed if all deadlines were checked. An example of the approach is as follows. Six tasks have the characteristics given in Table 11.16. The utilization of this task set is 0.965. The value of L is 59 and there are 34 deadlines that need to be checked in this interval using PDC.

Applying QPA results in just 14 points that need to be considered, and these correspond to the following values of t : 59, 53, 46, 43, 40, 33, 29, 24, 21, 19, 12, 9, 5 and 1. In other examples (Zhang and Burns, 2008), QPA typically requires only 1% of the effort of the original processor demand analysis scheme.

11.11.4 Blocking and EDF

When considering shared resources and blocking, there is a direct analogy between EDF and FPS. Where FPS suffers *priority inversion*, EDF suffers *deadline inversion*. This is when a task requires a resource that is currently locked by another task with a longer deadline. Not surprisingly inheritance and ceiling protocols have been developed for EDF but, as with earlier comparisons, the EDF schemes are somewhat more complex (Baruah, 2006).

As priorities are static, it is easy to determine which tasks can block the task currently being analysed. With EDF, this relationship is dynamic; it depends on which tasks (with longer deadlines) are active when the task is released. And this varies from one release to another throughout the hyper-period.

Probably the best scheme for EDF is the **Stack Resource Policy** (SRP) of Baker (1991). This works in a very similar way to the immediate ceiling priority protocol for FPS (indeed SRP influenced the development of ICPP). Each task, under SRP, is assigned a **preemption level**. *Preemption levels reflect the relative deadlines of the tasks, the shorter the deadline the higher the preemption level*; so they actually designate the static priority of the task as assigned by the deadline monotonic scheme. At run-time, resources are given ceiling values based on the maximum preemption level of the tasks that use the resource.

When a task is released, it can only preempt the currently executing task if its absolute deadline is shorter and its preemption level is higher than the highest ceiling of the currently locked resources.

The result of applying this protocol is identical to applying ICPP (on a single processor). Tasks suffer only a single block (it is as they are released), deadlocks are prevented and a simple formula is available for calculating the blocking time. The blocking term, once calculated, can be incorporated into PDC and QPA.

11.11.5 Aperiodic tasks and EDF execution-time servers

Following the development of server technology for fixed-priority systems, most of the common approaches have been reinterpreted within the context of dynamic EDF systems. For example there is a Dynamic Sporadic Server and a Dynamic Deferrable Server. Whereas the static system needs a priority to be assigned (which is done pre-run-time), the dynamic version needs to compute a deadline each time it needs to execute.

In addition to these common forms of servers there are also a number that are EDF-specific. These take the form of virtual (but slow) processors that can guarantee C in T . So an aperiodic task that requires to execute for $3C$ will be guaranteed to complete in $3T$ if the server has no other work to do. To find more about EDF servers and EDF scheduling in general, the reader is referred to the books by Liu and Buttazzo in the Further Reading section at the end of this chapter.

11.12 Dynamic systems and online analysis

Earlier in this chapter it was noted that there is a wide variety of scheduling schemes that have been developed for different application requirements. For hard real-time systems, offline analysis is desirable (indeed it is often mandatory). To undertake such analysis requires:

- arrival patterns of incoming work to be known and bounded (this leads to a fixed set of tasks with known periods or worst-case arrival intervals);
- bounded computation times;
- a scheduling scheme that leads to predictable execution of the application tasks.

This chapter has shown how fixed-priority scheduling (and to a certain extent, EDF) can provide a predictable execution environment.

In contrast to hard systems, there are dynamic soft real-time applications in which arrival patterns and computation times are not known *a priori*. Although some level of offline analysis may still be applicable, this can no longer be complete and hence some form of online analysis is required.

The main objective of an online scheduling scheme is to manage any overload that is likely to occur due to the dynamics of the system's environment. It was noted earlier that EDF is a dynamic scheme that is an optimal scheduling discipline. Unfortunately, EDF also has the property that during transient overloads it performs very badly. It is possible to get a cascade effect in which each task misses its deadline but uses sufficient resources to result in the next task also missing its deadline.

To counter this detrimental domino effect, many online schemes have two mechanisms:

- (1) an admissions control procedure that limits the number of tasks that are allowed to compete for the processors;
- (2) an EDF dispatching routine for those tasks that are admitted.

An ideal admissions control algorithm prevents the processor getting overloaded so that the EDF routine works effectively.

If some tasks are to be admitted, while others are rejected, the relative importance of each task must be known. This is usually achieved by assigning each task a *value*. Values can be classified as follows.

- **Static** – the task always has the same value whenever it is released.
- **Dynamic** – the task's value can only be computed at the time the task is released (because it is dependent on either environmental factors or the current state of the system).
- **Adaptive** – here the dynamic nature of the system is such that the value of the task will change during its execution.

To assign static values (or to construct the algorithm and define the input parameters for the dynamic or adaptive schemes) requires the domain specialists to articulate their understanding of the desirable behaviour of the system. As with other areas of computing, knowledge elicitation is not without its problems, but these issues will not be considered here (see Burns et al., 2000).

One of the fundamental problems with online analysis is the trade-off that has to be made between the quality of the scheduling decision and the resources and time needed to make the decision. At one extreme, every time a new task arrives, the complete set of tasks could be subject to an exact test such as those described in this chapter. If the task set is not schedulable, the lowest value task is dropped and the test repeated (until a schedulable set is obtained). This approach (which is known as **best-effort**) is optimal for static or dynamic value assignment – *but only if the overheads of the tests are ignored*. Once the overheads are factored in, the effectiveness of the approach is seriously compromised. In general, heuristics have to be used for online scheduling and

it is unlikely that any single approach will work for all applications. This is still an active research area. It is clear, however, that what is required is not a single policy defined in a language or OS standard, but mechanisms from which applications can program their own schemes to meet their particular requirements.

The final topic to consider in this section is **hybrid systems** that contain both hard and dynamic components. It is likely that these will become the norm in many application areas. Even in essentially static systems, value-added computations, in the form of soft or firm tasks that improve the quality of the hard tasks, are an attractive way of structuring systems. In these circumstances, as was noted in Section 11.6.1, the hard tasks must be protected from any overload induced by the behaviour of the non-hard tasks. One way of achieving this is to use FPS for the hard tasks and execution-time servers for the remaining work. The servers can be executed at a given priority level, but can embody whatever admissions policy is desirable and service the incoming dynamic work using EDF.

11.13 Worst-case execution time

In all the scheduling approaches described so far (that is, cyclic executives, FPS and EDF), it is assumed that the worst-case execution time of each task is known. This is the maximum any task invocation/release (i.e. job) could require.

Worst-case execution time estimation (represented by the symbol C but also known by the acronym WCET) can be obtained by either measurement or analysis. The problem with measurement is that it is difficult to be sure when the worst case has been observed. The drawback of analysis is that an effective model of the processor (including caches, pipelines, branch prediction, out-of-order execution, memory wait states and so on) must be available.

Most analysis techniques involve two distinct activities. The first takes the task and decomposes its code into a directed graph of **basic blocks**. These basic blocks represent straightline code. The second component of the analysis takes the machine code corresponding to a basic block and uses the processor model to estimate its worst-case execution time.

Once the times for all the basic blocks are known, the directed graph can be collapsed. For example, a simple choice construct between two basic blocks will be collapsed to a single value (that is, the largest of the two values for the alternative blocks). Loops are collapsed using knowledge about maximum bounds.

More sophisticated graph reduction techniques can be used if sufficient semantic information is available. To give just a simple example of this, consider the following code:

```
for I in 1.. 10 loop
  if Cond then
    -- basic block of cost 100
  else
    -- basic block of cost 10
  end if;
end loop;
```

With no further information, the total ‘cost’ of this construct would be $10 \times 100 +$ the cost of the loop construct itself, giving a total of, say, 1005 time units. It may, however, be possible to deduce (via static analysis of the code) that the condition `Cond` can only be true on at most three occasions. Hence a less pessimistic cost value would be 375 time units.

Other relationships within the code may reduce the number of feasible paths by eliminating those that cannot possibly occur; for instance, when the ‘if’ branch in one conditional statement precludes a later ‘else’ branch. Techniques that undertake this sort of semantic analysis usually require annotations to be added to the code. The graph reduction process can then make use of tools such as ILP (Integer Linear Programming) to produce a tight estimate of worst-case execution time. They can also advise on the input data needed to drive the program down the path that gives rise to this estimation.

Clearly, if a task is to be analysed for its worst-case execution time, the code itself needs to be restricted. For example, all loops and recursion must be bounded, otherwise it would be impossible to predict offline when the code terminates. Furthermore, the code generated by the compiler must also be analysable.

The biggest challenge facing worst-case execution time analysis comes from the use of modern processors with multicores, on-chip caches, pipelines, branch predictors and so on. All of these features aim to reduce *average* execution time, but their impact on *worst-case* behaviour can be hard to predict. If one ignores these features the resulting estimates can be very pessimistic, but to include them is not always straightforward. One approach is to assume non-preemptive execution, and hence all the benefits from caching and so on can be taken into account. At a later phase of the analysis, the number of actual preemptions is calculated and a penalty applied for the resulting cache misses and pipeline refills.

To model in detail the temporal behaviour of a modern processor is non-trivial and may need proprietary information that can be hard to obtain. For real-time systems one is left with the choice of either using simpler (but less powerful) processor architectures or putting more effort into measurement. Given that all high-integrity real-time systems will be subject to considerable testing, an approach that combines testing and measurement for code units (basic blocks) but path analysis for complete components seems appropriate with today’s technology.

This brief discussion has only addressed a few of the issues involved with WCET estimation. A comprehensive coverage would perhaps triple the size of this chapter. Interested readers are referred to the Further Reading section of this chapter.

11.14 Multiprocessor scheduling

All the analysis presented so far in this chapter has been concerned with the scheduling of concurrent tasks on to a single computer. As processors have become more powerful, the size and complexity of the applications that can fit on to a single processor have increased substantially. However, for a significant class of system, there is the need to move to a multiprocessor execution platform. This can take the form of a relatively simple dual processor or may involve a large multicore chip. The processors may all be of an identical type (homogeneous) or may have quite different characteristics (heterogeneous). Also the links between the processors may be a shared memory bus or an

Task	<i>T</i>	<i>D</i>	<i>C</i>
<i>a</i>	10	10	5
<i>b</i>	10	10	5
<i>c</i>	12	12	8

Table 11.17 Example task set.

independent network. Whatever the architecture, the scheduling problem for a multiprocessor system is significantly more complicated than the single processor case. Here three issues are addressed, the placement of tasks to processors, the scheduling of any shared network and the implementation of locks in shared-memory multiprocessor architectures.

11.14.1 Global or partitioned placement

The first new issue that must be addressed with a multiprocessor platform is placement. This is the mapping of tasks to processors. Two schemes are possible: **global** and **partitioned**. As the names imply, a partitioned scheme is a pre-run-time allocation; the dynamic global alternative allocates tasks as they become runnable, and even, during execution a task may move from one processor to another. Both placement schemes have their advantages and disadvantages. The partitioned approach has the benefit of not requiring any run-time support and of being able to cater for certain types of systems that the global scheme has difficulty with. Consider a simple three task system that is to be implemented on two identical processors (see Table 11.17).

The total utilization of the system is 1.66 so it is capable of being scheduled (as $1.66 < 2$). A simple global placement scheme using either fixed priority or EDF would allocate *a* and *b* to the two processors. They would then execute until time 5. Now *c* can be placed on either processor as they are both free, but there are only 7 units of time left before the deadline at time 12. Hence *c* will fail to meet its deadline.

The partitioned approach would allocate *a* and *b* to one processor (where they will completely utilize its capacity), leaving the other processor to *c* which can then easily meet its deadline. A different example, however, will show the benefits of the global scheme. Table 11.18 also has a three task / two processor configuration. No partitioned scheme can schedule this system. However, a global scheme that starts with *d* and *f* for one tick and then executes *d* and *e* for the next eight ticks will be able to run *f* and *e* for the 10th tick and satisfy all requirements.

Task	<i>T</i>	<i>D</i>	<i>C</i>
<i>d</i>	10	10	9
<i>e</i>	10	10	9
<i>f</i>	10	10	2

Table 11.18 A further example task set.

The only way to fit this example into a partitioned scheme is to split f into two identical length parts. This artificial decomposition of a software model was criticized earlier within the context of cyclic executives and cannot be recommended.

The main challenge for the partitioned approach is to generate a valid allocation. Once an allocation has been formed, it is straightforward to analyse each processor in turn using the analysis already available for single processors. Each processor can be scheduled via fixed priorities or EDF – it would even be possible to mix the schemes with EDF on some processors and FPS on others. An allocation must not ‘overfill’ any processor and hence task mapping is similar to the classic bin-packing problem. Optimal schemes for large numbers of tasks and processors are therefore not possible – heuristics must be employed.

The challenge for the global scheme is to identify the optimal scheduling policy. For a single processor, EDF is optimal, but no optimal scheme is known for general task models executing on multiprocessor platforms. The earlier example in Table 11.17 shows that EDF scheduling would lead to c missing its deadline, but the system is schedulable as evident from the partitioned approach. Not only are there no optimal schemes available, but what policies are available tend not to be sustainable – a schedulable system may become unschedulable if, for example, an extra processor is added. For these reasons, most current systems use a partitioned approach, which also has the advantage, as noted earlier, of run-time efficiency. Nevertheless, for the future, global schemes have the potential to deliver better schedulability.

Notwithstanding the lack of maturity in this area, it is possible to report some useful results. It was noted at the beginning of this chapter that single processor analysis started from a simple task model (in particular, tasks are independent and periodic, and have deadline equal to period). For uniprocessors, the model has been generalized significantly, but for multiprocessor scheduling, useful results are only really known for the simple task model. Here a number of such results are stated.

- (1) There is a global scheduling algorithm called *pfair* that is able to schedule any periodic system with utilization $\leq M$ on M identical processors (Baruah et al., 1996). However, schedules generated by the *pfair* algorithm tend to have a large number of preemptions, and tasks are frequently required to move from one processor to another.
- (2) For FP scheduling with partitioned placement, the following utilization-based sufficient schedulability test has been derived for a first-fit (using decreasing task utilization) placement strategy (Oh and Baker, 1998):

$$U \leq M(\sqrt{2} - 1) \quad (11.28)$$

- (3) A sufficient utilization-based schedulability test is also known for EDF with partitioned first-fit placement. Let U_{\max} denote the largest utilization of any task in the system: $U_{\max} = \max_{i=1}^N \left(\frac{C_i}{T_i} \right)$. Letting β denote $\lfloor 1/U_{\max} \rfloor$, this test is as follows (Lopez et al., 2004):

$$U \leq \frac{\beta M + 1}{\beta + 1} \quad (11.29)$$

- (4) A sufficient schedulability test is known for global EDF (Goossens et al., 2003), that depends only upon the utilization of the task system and U_{\max} .

$$U \leq M - (M - 1)U_{\max} \tag{11.30}$$

- (5) A variant of EDF with global placement, called fpEDF, has been proposed (Baruah, 2004) that
- (a) assigns greatest (fixed) priority to tasks with utilization $> \frac{1}{2}$ (if there are any such tasks), and
 - (b) schedules the remaining tasks according to EDF.

A couple of schedulability tests have been derived for fpEDF. One uses the utilization of the task system:

$$U \leq \left(\frac{M + 1}{2} \right) \tag{11.31}$$

Another, superior, test for fpEDF uses both the utilization of the system and the largest utilization of any task in the system:

$$U \leq \max \left[M - (M - 1)U_{\max}, \frac{M}{2} + U_{\max} \right] \tag{11.32}$$

A direct comparison between all these bounds would be misleading as the bounds are only sufficient. Tighter bounds are possible; indeed a number have been proposed in the literature for some of the above schemes. However, to illustrate the results that these test do provide, a couple of scenarios are defined and the tests applied. Case I has 10 processors ($M = 10$) with the highest utilization task having a capacity of 0.1. Case II has $M = 4$ and $U_{\max} = 2/3$. Table 11.19 gives the maximum utilization that is guaranteed to lead to a schedulable system for each of the scheduling schemes. To compute the average ‘per-processor’ utilization these values need to be divided by M (which is 10 in the first scenario and 4 in the other).

11.14.2 Scheduling the network

The next issue to consider is the communications infrastructure that links the different processors. For bus-based tightly coupled multiprocessors, the behaviour of multilevel

Scheme	Case I	Case II
Partitioned FP (Eqn 11.28)	4.14	1.66
Partitioned EDF (Eqn 11.29)	9.18	2.50
Global EDF (Eqn 11.30)	9.10	2.00
fpEDF (Eqn 11.32)	9.10	2.67

Table 11.19 Utilization bounds.

caches makes worst-case execution time analysis even harder for these platforms. Heterogeneous processors and hierarchical multispeed memories also add significantly to these difficulties. With network-based connections, the messages must themselves be scheduled if end-to-end data flows through the system are to be guaranteed. There are many different network protocols with some being more amenable to timing analysis than others. It is beyond the scope of this book to discuss these different protocols in detail, but two specific schemes are worth noting.

- **Time Division Multiple Access (TDMA)** – here each processor is allocated a fixed time slot within a specified cycle in which tasks hosted on that processor can generate messages.
- **Control Area Network (CAN)** – here each message is given a fixed priority and the network supports priority-based arbitration.

So with TDMA, which is only really applicable to static task allocation, no two processors ever wish to produce messages at the same time. With CAN, competition can occur but priorities are used to order the messages. Being priority-based, the standard RTA presented earlier in this chapter is directly applicable to scheduling CAN. As a message cannot be preempted once it has started to be transmitted, the non-preemptive form of the analysis is the one employed with CAN (see Section 11.10.3), where the parameter C is now the time needed to transmit the message.

The use of a network within a system opens up a number of issues as well as message scheduling. Unless the hardware architecture is fully connected, routing needs to be addressed. Static and dynamic route-finding are possible. Fault tolerance over message corruption is normally dealt with by the transmission protocol, but extra messages and perhaps alternative routes may be employed and these must be accommodated into the scheduling analysis.

11.14.3 Mutual exclusion on multiprocessor platforms

The final issue to address in this short review of the major problems involved in scheduling multiprocessor platforms is the provision of mutual exclusion over shared objects. In networked systems, these objects are typically controlled by a single thread so there are no new problems to solve, but for shared memory systems there is now the need to provide protection from true parallel access. The priority inheritance and priority ceiling protocols no longer work as they depend on an executing high-priority task preventing a lower-priority task from executing. This clearly will not occur if the lower-priority task is on another processor. There are no simple equivalent protocols to those for single processors.

To implement mutual exclusion in a multiprocessor shared memory system usually requires locks that can be contended for globally. When a task holds such a lock it is usual to prevent it from being preempted locally (as this would further delay tasks on other processors waiting for the lock). When a lock is requested but not granted, the task will typically busy-wait on the lock, waiting for it to become free – this is known as **spinning**. Obviously this spinning time will add to the task's execution time and hence has to be bounded if the task's interference on lower-priority tasks is to be calculated. A system with many globally shared objects and nested usage patterns (i.e. accessing one

object whilst holding the lock on others) will be harder to analyse and the analysis itself is likely to be pessimistic. Also error conditions such as deadlocks and livelocks are now possible whereas they were prevented by some of the single processor protocols.

Because of these difficulties with global locks, the use of lock-free algorithms is attractive. Here multiple copies of the shared object are supported and if necessary actions are repeated if conflicts over the copies have occurred. To give a simple example, consider an object that is read by many tasks but updated by only one. While the update is happening all reads are made to an old copy of the object. Once the update is finished a single flag is set to make the new copy available to future read operations. If the timing constraints on the system allow concurrent reads and writes then it must be acceptable for the read operation to get the old value – if it had arrived any earlier (or the writing task any later) then the old copy would have been the ‘current’ one.

Overall, multiprocessor systems whilst providing more computational power introduce a number of challenges for real-time systems. The move from scheduling a single resource to the coordination of multi-resources is a major one that requires a holistic approach to system scheduling. However, the core of this approach will always be the management of each individual resource.

11.15 Scheduling for power-aware systems

All of the scheduling results presented in this chapter have the common form of: given a set of execution time requirements (the C s), will all the tasks complete by their deadlines (the D s)? This assumes a fixed speed processor (or processors) so that the worst-case execution time values can be obtained prior to attempting the system-wide scheduling analysis. There are, however, variable speed processors that can give rise to a difference scheduling question – at what speed must the processor execute in order for the tasks to be schedulable?

Variable speed resources are typically found in power-aware applications, that is in embedded systems that run on batteries. Examples of such systems are mobile devices and nodes in a sensor net. All battery-based systems have the need to preserve energy and thereby extend their operational life or periods between recharges.

To save power, the voltage to the processor is reduced with the result that it runs slower. But the saving is non-linear. Halving the speed of a processor may quadruple its life. Some processors have variable speed, others support just a finite set of speed settings. From the point of view of the scheduling analysis, the verification problem now has two stages.

- With the processor running at its maximum speed (Max), is the system schedulable? This is a standard test.
- If the system is schedulable, by what maximum factor k can all the C values be increased so that the system remains schedulable?

There is no simple way to compute k , rather it needs to be found by a branch and bound search. Consider the fixed priority example given in Table 11.20 where the C values are those that are appropriate for the maximum speed of the processor.

Task	<i>T</i>	<i>D</i>	<i>C</i>
<i>a</i>	70	70	5
<i>b</i>	120	100	7
<i>c</i>	200	200	11

Table 11.20 Example task set with maximum speed.

Task	<i>T</i>	<i>D</i>	<i>C</i>	<i>R</i>
<i>a</i>	70	70	25	25
<i>b</i>	120	100	35	60
<i>c</i>	200	200	55	200

Table 11.21 Example task set with reduced speed.

This is clearly schedulable ($R_a = 5$, $R_b = 12$ and $R_c = 23$). If k is given the value 10 (i.e. execution times are now 50, 70 and 110) then the utilization is greater than 1 so the system is clearly unschedulable. So k must lie between 1 and 10. Using, for illustration, only integer values for k , the value 6 could be tried next (result unschedulable) then 4 (schedulable) and then 5. The results for $k = 5$ are shown in Table 11.21. Note the response time for task c is just on its deadline (200); any increase in any C parameter would cause this task to become unschedulable. Hence $k = 5$ is the optimum value and it is possible to conclude that the task set is schedulable on a processor with speed $\text{Max}/5$.

This discussion has focused on statically fixing the processor speed so that all deadlines are (just) met. In more dynamic systems where the work load fluctuates at run-time it is possible to change the processor’s speed whilst continuing to execute the application. For all processors there is a cost (overhead) in making these changes. It is also necessary to take into account the impact on memory performance and I/O devices – savings in processing cost may not lead to overall system economy.

11.16 Incorporating system overheads

In all the analysis presented so far in this chapter, the overheads of actually implementing the multitasking system software have been ignored. Clearly for a real system this is not acceptable and hence the scheduling equations need to be expanded to include terms for the overhead factors. The following characteristics are typical of many operating system kernels or language run-time support systems.

- The cost of a context switch between tasks is not negligible and may not be a single value. The cost of a context switch to a higher-priority periodic task (following, for example, a clock interrupt) may be higher than a context switch from a task to a lower-priority task (at the end of the high-priority task’s execution). For systems with a large number of periodic tasks, an additional cost will be incurred for

manipulating the delay queue (for periodic tasks when they execute, say, an Ada 'delay until' statement).

- All context switch operations are non preemptive.
- The cost of handling an interrupt (other than the clock) and releasing an application sporadic task is not insignificant. Furthermore, for DMA and channel-program controlled devices, the impact of shared-memory access can have a non-trivial impact on worst-case performance – such devices are best avoided in hard real-time systems.
- A clock interrupt (say every 10 ms) could result in periodic tasks being moved from a delay queue to the dispatch/ready queue. The cost for this operation varies depending on the number of tasks to be moved.

In addition to the above, the scheduling analysis must take into account the features of the underlying hardware, such as the impact of the cache and pipeline.

11.16.1 Modelling non-trivial context switch times

Most scheduling models ignore context switch times. This approach is, however, too simplistic if the total cost of the context switches is not trivial when compared with the application's own code. Figure 11.12 illustrates a number of significant events in the execution of a typical periodic task.

A – the clock interrupt that designates the notional time at which the task should start (assuming no release jitter or non-preemptive delay – if the interrupts were disabled due to the operation of the context switch then the clock handler would have its execution delayed; this is taken into account in the scheduling equations by the blocking factor B).

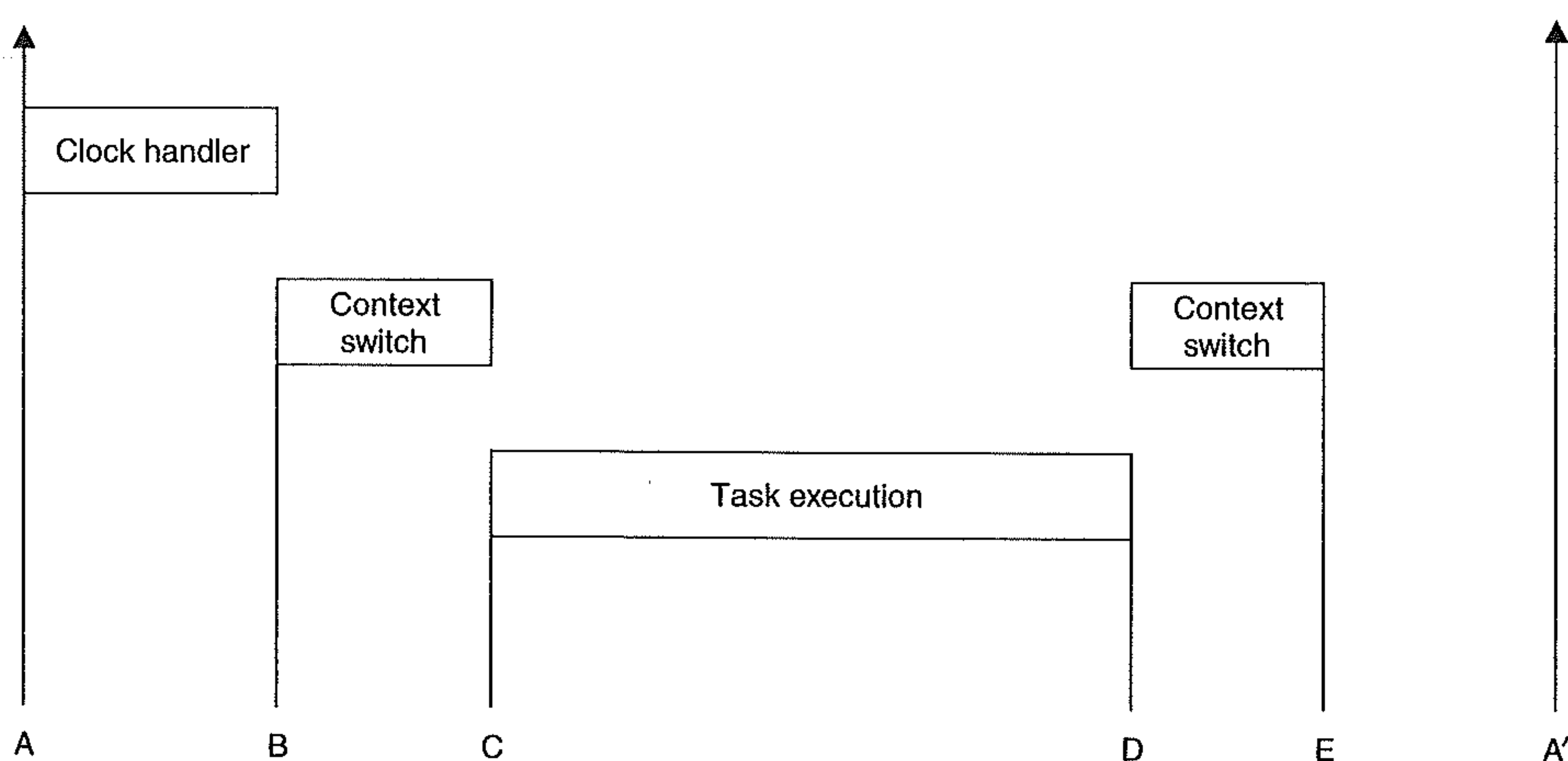


Figure 11.12 Overheads when executing tasks.

- B – the earliest time that the clock handler can complete; this signifies the start of the context switch to the task (assume it is the highest priority runnable task).
- C – the actual start of the execution of the task.
- D – the completion of the task (the task may be preempted a number of times between C and D).
- E – the completion of the context switch away from the task.
- A' – the next release of the task.

The typical requirement for this task is that it completes before its next release (that is, $D < A'$), or before some deadline prior to its next release. Either way, D is the significant time, not E. Another form of requirement puts a bound on the time between the start of execution and termination (that is, $D - C$). This occurs when the first action is an input and the last an output (and there is a deadline requirement between the two). While these factors affect the meaning of the task's own deadline (and hence its response time) they do not affect the interference this task has on lower-priority tasks; here the full cost of both context switches counts. Recall that the basic scheduling equation (11.7) has the form:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

This now becomes (for periodic tasks only):

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j) \quad (11.33)$$

where CS^1 is the cost of the initial context switch (to the task) and CS^2 is the cost of the context switch away from each task at the end of its execution. The cost of putting the task into the delay queue (if it is periodic) is incorporated into C_i . Note that in practice this value may depend on the size of the queue; a maximum value would need to be incorporated into C_i .

This measure of the response time is from point B in Figure 11.12. To measure from point C, the first CS^1 term is removed. To measure from point A (the notional true release time of the task) requires the clock behaviour to be measured (see Section 11.16.3).

For multiprocessor systems the context switch itself may be more complicated if global placement is used as tasks may need to migrate from one processor to another. This will add to the context switch overhead and will make the prediction of worst-case execution time (WCET) more difficult as the cache may not be shared between all the processors.

11.16.2 Modelling sporadic tasks

For sporadic tasks released by other sporadic tasks, or by periodic tasks, Equation (11.33) is a valid model of behaviour. However, the computation time for the task, C_i , must include the overheads of blocking on the agent that controls its release.

When sporadic tasks are released by an interrupt, priority inversion can occur. Even if the sporadic has a low priority (due to its having a long deadline) the interrupt itself will be executed at a high hardware priority level. Let Γ_s be the set of sporadic tasks released by interrupts. Each interrupt source will be assumed to have the same arrival characteristics as the sporadic that it releases. The additional interference these interrupt handlers have on each application task is given by:

$$\sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil IH$$

where IH is the cost of handling the interrupt (and returning to the running task, having released the sporadic task).

This representation assumes that all interrupt handlers give rise to the same cost; if this is not the case then IH must be defined for each k . Equation (11.33) now becomes:

$$\begin{aligned} R_i = & CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j) \\ & + \sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil IH \end{aligned} \tag{11.34}$$

Within Ada, timing events are also used to release sporadic tasks or to undertake short event-handling activities. As timing events are similar to interrupts they can be modelled in the way shown above.

11.16.3 Modelling the real-time clock handler

To support periodic tasks, the execution environment must have access to a real-time clock that will generate interrupts at appropriate times. An ideal system will use an interval timer, and will interrupt only when a periodic task needs to be released. The more common approach, however, is one in which the clock interrupts at a regular rate (say once every 10 ms) and the handler must decide if none, one or a number of periodic tasks must be released. The ideal approach can be modelled in an identical way to that introduced for sporadic tasks (see Section 11.16.2). With the regular clock method, it is necessary to develop a more detailed model as the execution times of the clock handler can vary considerably. Table 11.22 gives possible times for this handler (for a clock period of 10 ms). Note that if the worst case was assumed to occur on all occasions, over

Queue state	Clock handling time, μs
No tasks on queue	16
Tasks on queue but none removed	24
One task removed	88
Two tasks removed	128
Twenty-five tasks removed	1048

Table 11.22 Clock handling overheads.

10% of the processor would have to be assigned to the clock handler. Moreover, all this computation occurs at a high (highest) hardware priority level, and hence considerable priority inversion is occurring. For example, with the figures given in the table, at the LCM (least common multiple) of the 25 periodic tasks $1048 \mu s$ of interference would be suffered by the highest priority application task that was released. If the task was released on its own then only $88 \mu s$ would be suffered. The time interval is represented by B–A in Figure 11.12.

In general, the cost of moving N periodic tasks from the delay queue to the dispatch queue can be represented by the following formula:

$$C_{clk} = CT^c + CT^s + (N - 1)CT^m$$

where CT^c is the constant cost (assuming there is always at least one task on the delay queue), CT^s is the cost of making a single move, and CT^m is the cost of each subsequent move. This model is appropriate due to the observation that the cost of moving just one task is often high when compared with the additional cost of moving extra tasks. With the kernel considered here, these costs were:

CT^c	$24 \mu s$
CT^s	$64 \mu s$
CT^m	$40 \mu s$

To reduce the pessimism of assuming that a computational cost of C_{clk} is consumed on each execution of the clock handler, this load can be spread over a number of clock ticks. This is valid if the shortest period of any application task, T_{\min} is greater than the clock period, T_{clk} . Let M be defined by:

$$M = \left\lceil \frac{T_{\min}}{T_{clk}} \right\rceil$$

If M is greater than 1 then the load from the clock handler can be spread over M executions. In this situation, the clock handler is modelled as a task with period T_{\min} and computation time C'_{clk} :

$$C'_{clk} = M(CT^c + CT^s) + (N - M)CT^m$$

This assumes $M \leq N$.

Equation (11.34) now becomes

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j) + \sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil IH + \left\lceil \frac{R_i}{T_{\min}} \right\rceil C'_{clk} \quad (11.35)$$

To give further improvements (to the model) requires a more exact representation of the clock handler's actual execution. For example, using just CT^c and CT^s the following

equation can easily be derived:

$$\begin{aligned}
 R_i = & CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j) \\
 & + \sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil IH + \left\lceil \frac{R_i}{T_{clk}} \right\rceil CT_c \\
 & + \sum_{g \in \Gamma_p} \left\lceil \frac{R_i}{T_g} \right\rceil CT_s
 \end{aligned} \tag{11.36}$$

where Γ_p is the set of periodic tasks.

It is left as an exercise for the reader to incorporate the three-parameter model of clock handling (see Exercise 11.16).

Summary

A scheduling scheme has two facets: it defines an algorithm for resource sharing and a means of predicting the worst-case behaviour of an application when that form of resource sharing is used.

Many current periodic real-time systems are implemented using a cyclic executive. With this approach, the application code must be packed into a fixed number of 'minor cycles' such that the cyclic execution of the sequence of minor cycles (called a 'major cycle') will enable all system deadlines to be met. Although an effective implementation strategy for small systems, there are a number of drawbacks with this cyclic approach.

- The packing of the minor cycles becomes increasingly difficult as the system grows.
- Sporadic activities are difficult to accommodate.
- Tasks with long periods (that is, longer than the major cycle) are supported inefficiently.
- Tasks with large computation times must be split up so that they can be packed into a series of minor cycles.
- The structure of the cyclic executive makes it very difficult to alter to accommodate changing requirements.

Because of these difficulties, this chapter has focused on the use of more dynamic scheduling schemes. The main topic of the chapter has been the fixed priority scheduling of a collection of tasks on a single processor. Following the description of a simple utilization-based test (which is only applicable to a restricted task model), the response time calculations were derived for a more flexible model. This model can accommodate sporadic tasks, task interactions, non-preemptive sections, release jitter, aperiodic servers, fault-tolerant systems and an arbitrary relationship between a task deadline (D) and its minimum arrival interval (T).

Intertask synchronization, such as mutual exclusive access to shared data, can give rise to priority inversion unless some form of priority inheritance is used. Two particular protocols were described in detail in this chapter: 'original ceiling priority inheritance' and 'immediate ceiling priority inheritance'.

With priority-based scheduling, it is important that the priorities are assigned to reflect the temporal characteristic of the task load. Three algorithms have been described in this chapter:

- rate monotonic – for $D = T$
- deadline monotonic – for $D \leq T$
- arbitrary – for $D > T$.

The other significant scheduling approach is EDF – here the task with the shortest (earliest) deadline is the one to execute first. EDF has the advantage that it optimally allocates the processor. If an application cannot be scheduled by EDF then it cannot be scheduled by any other approach. For simple systems with $D = T$ then a very simple and exact utilization-based test can be used to check schedulability. When $D \leq T$ then Processor Demand Analysis has to be undertaken. An efficient approach to this form of analysis called QPA was introduced and illustrated by examples.

Also in this chapter a number of other scheduling topics have been referred to (even if they have not been dealt with comprehensively). Amongst these are: worst-case execution time estimation, multiprocessor platforms, dynamic open systems and power-aware applications. Finally attention was focused on the overheads of implementing multitasking application code. It was shown how the standard response time equations can be extended to incorporate parameters to characterize the implementation's run-time behaviour.

Further reading

- Buttazzo, G. C. (1997) *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. New York: Kluwer Academic.
- Davis, R., Zaboos, A. and Burns, A. (2008) Effective exact schedulability tests for fixed priority real-time systems. *IEEE Transactions on Computing*, **57**(9), 1261–1276.
- Ermedahl, A. and Engblom, J. (2007) Execution time analysis for embedded real-time systems, in *Handbook of Real-Time and Embedded Systems*, I. Lee, J. Y-T. Leung and S. H. Son (eds). Boca Raton, FL: Chapman and Hall/CRC.
- Klein, M. H., Ralya, T., Pollak, B., Obenza, R. and Gonzalez Harbour, M. (1993) *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. New York: Kluwer Academic.
- Liu, J. W. S (2000) *Real-Time Systems*. New York: Prentice Hall.
- Natarajan, S. (ed.) (1995) *Imprecise and Approximate Computation*. New York: Kluwer Academic.
- Rajkumar, R. (1993) *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. New York: Kluwer Academic.

- Sha, L. et al. (2004) Real time scheduling theory: A historical perspective. *Real-Time Systems*, **28**(2 – 3), 101–155.
- Stankovic, J. A., Spuni, M., Ramamritham, K. and Buttazzo, G. C. (1998) *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. New York: Kluwer Academic.

Exercises

- 11.1** Three logical tasks P , Q and S have the following characteristics. P : period 3, required execution time 1; Q : period 6, required execution time 2; S : period 18, required execution time 5.
Show how these tasks can be scheduled using the rate monotonic scheduling algorithm.
Show how a cyclic executive could be constructed to implement the three logical tasks.
- 11.2** Consider three tasks P , Q and S . P has a period of 100 ms in which it requires 30 ms of tasking. The corresponding values for Q and S are (5,1) and (25,5) respectively. Assume that P is the most important task in the system, followed by S and then Q .
- (1) What is the behaviour of the scheduler if priority is based on importance?
 - (2) What is the processor utilization of P , Q and S ?
 - (3) How should the tasks be scheduled so that all deadlines are met?
 - (4) Illustrate one of the schemes that allows these tasks to be scheduled.
- 11.3** To the above task set is added a fourth task (R). Failure of this task will not lead to safety being undermined. R has a period of 50 ms, but has a processing requirement that is data dependent and varies from 5 to 25 ms. Discuss how this task should be integrated with P , Q and S .
- 11.4** Figure 11.13 illustrates the behaviour of four periodic tasks w , x , y and z . These tasks have priorities determined by the rate monotonic scheme, with the result that $\text{priority}(w) > \text{priority}(x) > \text{priority}(y) > \text{priority}(z)$. Each task's period starts at time S and terminates at T . The four tasks share two resources that are protected by binary semaphores A and B . On the diagram the tag A (and B) implies 'do a wait operation on the semaphore'; the tag A' (and B') implies 'do a signal operation on the semaphore'. Table 11.23 summarizes the task's requirements.
The figure shows the execution histories of the four tasks using static priorities. For example, x starts at time 2, executes a successful wait operation on B at time 3 but unsuccessfully waits on A at time 4 (z has already locked A). At time 13 it executes again (that is, it now has lock on A), it releases A at time 14 and B at time 15. It is now preempted by w , but executes again at time 16. Finally it terminates at time 17.
Redraw Figure 11.13 to illustrate the behaviour of these tasks if priority inheritance is employed.

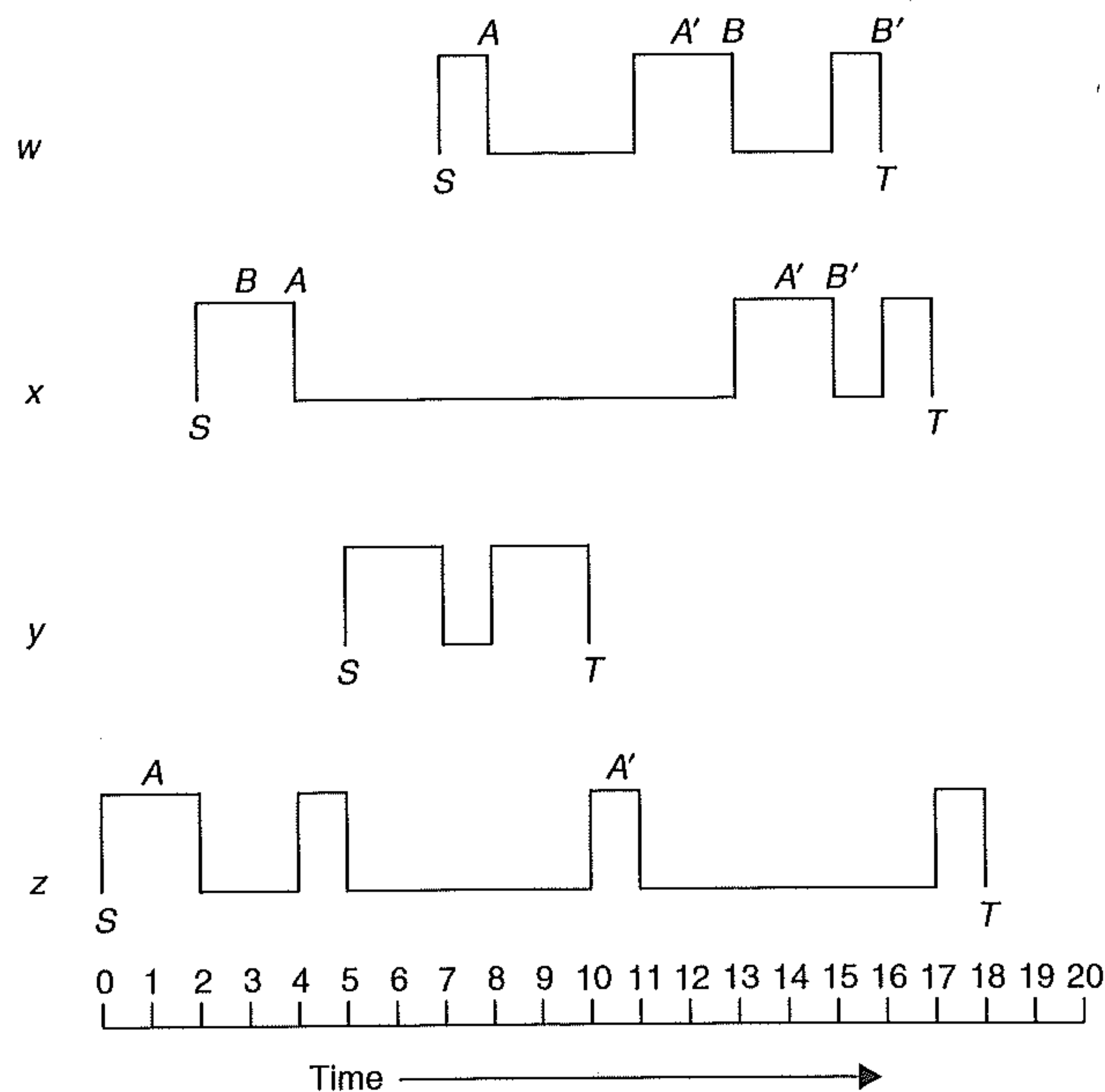


Figure 11.13 The behaviour of four periodic tasks in Exercise 11.4.

Task	Priority	Start time	Required processor time	Semaphores used
w	10	7	4	A,B
x	8	2	5	A,B
y	6	5	4	—
z	4	0	5	A

Table 11.23 Summary of the task’s requirements for Exercise 11.4.

- 11.5 Redraw the figure given in Exercise 11.4 to illustrate the behaviour of these tasks if immediate priority ceiling inheritance is employed.
- 11.6 With the priority ceiling protocol, it is possible to calculate the maximum time any task can be blocked by the operation of a lower-priority task. What is the rule for calculating this blocking? Illustrate the answer by calculating the maximum blocking time for each task in the following example. A program consists of five tasks, *a*, *b*, *c*, *d*, *e* (these are listed in priority order with *a* having the highest priority), and six resources *R*1, . . . , *R*6 (protected by semaphores implementing the priority ceiling protocol). The resource accesses have worst case execution times given in Table 11.24.

<i>R</i> 1	<i>R</i> 2	<i>R</i> 3	<i>R</i> 4	<i>R</i> 5	<i>R</i> 6
50 ms	150 ms	75 ms	300 ms	250 ms	175 ms

Table 11.24 Summary of resource access times for Exercise 11.6.

Resources are used by the tasks according to Table 11.25.

Task	Uses
<i>a</i>	<i>R3</i>
<i>b</i>	<i>R1, R2</i>
<i>c</i>	<i>R3, R4, R5</i>
<i>d</i>	<i>R1, R5, R6</i>
<i>e</i>	<i>R2, R6</i>

Table 11.25 Summary of the tasks’ resource requirements for Exercise 11.6.

11.7 Is the task set shown in Table 11.26 schedulable using the simple utilization-based test given in Equation (11.1)? Is the task set schedulable using the response time analysis?

Task	Period	Execution time
<i>a</i>	50	10
<i>b</i>	40	10
<i>c</i>	30	9

Table 11.26 Summary of the tasks’ attributes for Exercise 11.7.

11.8 The task set shown in Table 11.27 is not schedulable using Equation (11.1) because *a* must be given the top priority due to its criticality. How can the task set be transformed so that it is schedulable? Note that the computations represented by *a* must still be given top priority.

Task	Period	Execution time	Criticality
<i>a</i>	60	10	HIGH
<i>b</i>	10	3	LOW
<i>c</i>	8	2	LOW

Table 11.27 Summary of the tasks’ attributes for Exercise 11.8.

11.9 The task set given in Table 11.28 is not schedulable using Equation (11.1), but does meet all deadlines when scheduled using fixed priorities. Explain why.

Task	Period	Execution time
<i>a</i>	75	35
<i>b</i>	40	10
<i>c</i>	20	5

Table 11.28 Summary of the tasks’ attributes for Exercise 11.9.

- 11.10** In Section 11.6, a sporadic task was defined as having a minimum inter-arrival time. Often sporadic tasks come in bursts. Update Equation (11.4) to cope with a burst of sporadic activities such that N invocations can appear arbitrarily close together in a period of T .
- 11.11** Extend the answer given above to cope with sporadic activity which arrives in bursts, where there may be N invocations in a period of T and each invocation must be separated by at least M time units.
- 11.12** To what extent can the response time equations given in this chapter be applied to resources other than the CPU? For example, can the equations be used to schedule access to a disk?
- 11.13** In a safety-critical real-time system, a collection of tasks can be used to monitor key environmental events. Typically, there will be a deadline defined between the event occurring and some output (which is in response to the event) being produced. Describe how periodic tasks can be used to monitor such events.
- 11.14** Consider the list of events (shown in Table 11.29) together with the computation costs of responding to each event. If a separate task is used for each event (and these tasks are implemented by preemptive priority-based scheduling) describe how Rate Monotonic Analysis can be applied to make sure all deadlines are met.

EVENT	Deadline	Computation time
A_Event	36	2
B_Event	24	1
C_Event	10	1
D_Event	48	4
E_Event	12	1

Table 11.29 Summary of events for Exercise 11.14.

- 11.15** How can the task set shown in Table 11.30 be optimally scheduled (using fixed-priority scheduling)? Is this task set schedulable?

Task	T	C	B	D
a	8	4	2	8
b	10	2	2	5
c	30	5	2	30

Table 11.30 Summary of tasks for Exercise 11.15.

- 11.16** Develop a model of clock handling which incorporates the three parameters CT^c , CT^s and CT^m (see Section 11.16.3).
- 11.17** Rather than using a clock interrupt to schedule periodic tasks, what would be the ramifications of only having access to a real-time clock?
- 11.18** A periodic task of period 40 ms is controlled by a clock interrupt that has a granularity of 30 ms. How can the worst-case response time of this task be calculated?

Chapter 12

Programming schedulable systems

12.1	Programming cyclic executives	12.6	C/Real-Time POSIX and fixed-priority scheduling
12.2	Programming preemptive priority-based systems	12.7	Real-Time Java and fixed-priority scheduling
12.3	Ada and fixed-priority scheduling	12.8	Programming EDF systems
12.4	The Ada Ravenscar profile	12.9	Mixed scheduling
12.5	Dynamic priorities and other Ada facilities		Summary
			Further reading
			Exercises

In the previous chapter a number of approaches to scheduling were introduced. These approaches were chosen partly due to their inherent value and maturity, but also because it is possible to use these techniques in programming real systems with current languages and operating systems. This chapter starts by looking briefly at the programming of cyclic executives, and then looks in detail at the support available for priority-based scheduling. EDF is also covered.

12.1 Programming cyclic executives

To implement a simple cyclic executive requires no special language support. None of the temporal or scheduling information needs to be represented in the application's code. Rather, a series of procedure calls are linked together with a simple hardware interrupt that is configured to occur at regular intervals. For example, the code for the example given in the previous chapter (see Section 11.1) would have the following simple form (with the interrupt occurring every 25 ms):

```
loop -- MAJOR CYCLE
  wait_for_interrupt;
  -- minor cycle 1
  procedure_for_a;
  procedure_for_b;
  procedure_for_c;
```

```

wait_for_interrupt;
-- minor cycle 2
procedure_for_a;
procedure_for_b;
procedure_for_d;
procedure_for_e;
wait_for_interrupt;
-- minor cycle 3
procedure_for_a;
procedure_for_b;
procedure_for_c;
wait_for_interrupt;
-- minor cycle 4
procedure_for_a;
procedure_for_b;
procedure_for_d;
end loop;

```

Some protection can be added to this structure to identify an execution time overrun. The last procedure in any minor cycle sets a binary flag to 1. The interrupt handler checks the value of the flag before it switches to the next minor cycle (and assigning 0 to the flag). If there is a minor cycle overflow then recovery can be attempted, or extra time provided, or the system restarted. Of course the recovery code cannot know which procedure in the minor cycle caused the overflow. There is no firewall protection between the procedures.

12.2 Programming preemptive priority-based systems

Few programming languages explicitly define priorities as part of their concurrency facilities. Those that do often provide only a rudimentary model.

One language that does attempt to give a more complete provision is Ada – this is, therefore, discussed in detail in the next section. Traditionally, priority-based scheduling has been more an issue for operating systems than languages. Hence, after a discussion of Ada, the facilities of C/Real-Time POSIX are reviewed. Ada and C/Real-Time POSIX assume that any schedulability analysis has been performed offline. More recently, Real-Time Java has attempted to provide the same facilities as Ada and C/Real-Time POSIX but with the option of supporting online feasibility analysis. This is considered in Section 12.7.

It should, perhaps, be noted at this point that although most general-purpose operating systems provide the notion of process or thread priority, their facilities are often inadequate for hard real-time programming. What is minimally required is:

- an effective range of priorities;
- immediate preemptive switching to higher priority processes when they become runnable;
- support for at least priority inheritance, but ideally some form of priority ceiling protocol.

12.3 Ada and fixed-priority scheduling

As indicated in the Preface, Ada is defined as a core language plus a number of annexes for specialized application domains. These annexes do not contain any new language features (in the way of new syntax) but define pragmas and library packages that must be supported if that particular annex is to be adhered to. This section considers some of the provisions of the Real-Time Systems Annex, in particular, those that allow priorities to be assigned to tasks (and protected objects).¹

To indicate that priority-based scheduling is required of the run-time implementation of the application's Ada program, the dispatching policy must be defined. This is done using a pragma:

```
pragma Task_Dispatching_Policy(FIFO_Within_Priority);
```

Where tasks share the same priority, then they are queued in FIFO order. Hence, as tasks become runnable, they are placed at the *back* of a notional **ready queue** for that priority level. One exception to this case is when a task is preempted; here the task is placed at the *front* of the ready queue for that priority level. On a multiprocessor system, it is implementation defined whether this policy is on a per-processor basis or across the entire processor cluster.

To allocate a priority to a task there are, within package `System`, the following declarations:

```
subtype Any_Priority is Integer range
    <implementation-defined>;
subtype Priority is Any_Priority range
    Any_Priority'First .. <implementation-defined>;
subtype Interrupt_Priority is Any_Priority range
    Priority'Last+1 .. Any_Priority'Last;
```

```
Default_Priority : constant Priority :=
    (Priority'First + Priority'Last)/2;
```

An integer range is split between standard priorities and (the higher) interrupt priority range. An implementation must support a range for `System.Priority` of at least 30 values and at least one distinct `System.Interrupt_Priority` value.

A task has its initial priority set by including a pragma in its specification:

```
task Controller is
    pragma Priority(10);
end Controller;
```

If a task-type definition contains such a pragma, then all tasks of that type will have the same priority unless a discriminant is used:

```
task type Servers(Task_Priority : System.Priority) is
    entry Service1(...);
    entry Service2(...);
```

¹Priority order can also be assigned to entry queues and the operation of the `select` statement. This section will, however, focus on task priorities and protected object ceiling priorities.

```

    pragma Priority(Task_Priority);
end Servers;

```

For protected objects acting as interrupt handlers, a special pragma is defined:

```
pragma Interrupt_Priority(Expression);
```

or simply

```
pragma Interrupt_Priority;
```

The definition, and use, of a different pragma for interrupt levels improves the readability of programs and helps to remove errors that can occur if task and interrupt priority levels are confused. However, the expression used in `Interrupt_Priority` evaluates down to `Any_Priority`, and hence it is possible to give a relatively low priority to an interrupt handler. If the expression is actually missing, the highest possible priority is assigned (i.e. `Any_Priority'Last`).

A priority assigned using one of these pragmas is called a **base priority**. A task may also have an **active priority** that may be higher where the `FIFO_Within_Priority` dispatching policy is being used – this will be explained in due course.

The main program, which is executed by a notional environmental task, can have its priority set by placing the `Priority` pragma in the main subprogram. If this is not done, the default value, defined in `System`, is used. Any other task that fails to use the pragma has a default base priority equal to the base priority of the task that created it.

In order to make use of the immediate ceiling priority protocol (ICPP), an Ada program must include the following pragma:

```
pragma Locking_Policy(Ceiling_Locking);
```

An implementation may define other locking policies; only `Ceiling_Locking` is required by the Real-Time Systems Annex. The default policy, if the pragma is missing, is implementation defined. To specify the ceiling priority for each protected object, the `Priority` and `Interrupt_Priority` pragmas defined earlier are used. If the pragma is missing, a ceiling of `System.Priority'Last` is assumed.

The exception `Program_Error` is raised if a task calls a protected object with a priority greater than the defined ceiling. If such a call were allowed, then this could result in the mutually exclusive protection of the object being violated. If it is an interrupt handler that calls in with an inappropriate priority, then the program becomes erroneous. This must ultimately be prevented through adequate testing and/or static analysis of the program.

With `Ceiling_Locking`, an effective implementation will use the thread of the calling task to execute not only the code of the protected call, but also the code of any other task that happens to have been released by the actions of the original call. For example, consider the following simple protected object:

```

protected Gate_Control is
  pragma Priority(28);
  entry Stop_And_Close;
  procedure Open;

```



```

private
  Gate: Boolean := False;
end Gate_Control;

protected body Gate_Control is
  entry Stop_And_Close when Gate is
  begin
    Gate := False;
  end Stop_And_Close;

  procedure Open is
  begin
    Gate := True;
  end Open;
end Gate_Control;

```

Assume a task T, priority 20, calls `Stop_And_Close` and is blocked. Later, task S (priority 27) calls `Open`. The thread that implements S will undertake the following actions.

- (1) Execute the code of `Open` for S.
- (2) Evaluate the barrier on the entry and note that T can now proceed.
- (3) Execute the code of `Stop_And_Close` for T.
- (4) Evaluate the barrier again.
- (5) Continue with the execution of S after its call on the protected object.

As a result, there has been no context switch. The alternative is for S to make T runnable at point (2); T now has a higher priority (28) than S (27) and hence the system must switch to T to complete its execution within `Gate_Control`. As T leaves, a switch back to S is required. This is much more expensive.

As a task enters a protected object, its priority may rise above the base priority level defined by the `Priority` or `Interrupt_Priority` pragmas. The priority used to determine the order of dispatching is the **active priority** of a task. This active priority is, when the dispatching policy is `FIFO_Within_Priority`, the maximum of the task's base priority and any priority it has inherited.

The use of a protected object is one way in which a task can inherit a higher active priority. There are others, for example:

- During activation – a task will inherit the active priority of the parent task that created it; remember (from Section 4.4) the parent task is blocked waiting for its child task to complete, and this could be a source of priority inversion without this inheritance rule.
- During a rendezvous – the task executing the **accept** statement will inherit the active priority of the task making the entry call (if it is greater than its own priority).

Note that the last case does not necessarily remove all possible cases of priority inversion. Consider a server task, S, with entry E and base priority L (low). A high-priority task makes a call on E. Once the rendezvous has started, S will execute with the higher

priority, but before *S* reaches the `accept` statement for *E*, it will execute with priority *L* (even though the high-priority task is blocked). This, and other candidates for priority inheritance, can be supported by an implementation. The implementation must, however, provide a pragma that the user can employ to select the additional conditions explicitly.

The Real-Time Systems Annex attempts to provide usable, flexible but extensible features. Clearly, this is not easy. Ada 83 suffered from being too prescriptive. However, the lack of a defined dispatching policy would be unfortunate, as it would not assist software development or portability. Hence Ada 95 defined a single policy, `FIFO_Within_Priority`, in the Annex. Ada 2005 has added to this policy by defining a number of others – some of these are described later in this chapter. First, however, means of restricting the facilities available to the programmer are considered.

12.4 The Ada Ravenscar profile

The earlier chapters of this book have demonstrated the extensive set of facilities that Ada provides for the support of real-time and concurrent programming. The expressive power of the full language is clearly extensive. There are, however, situations in which a restricted set of features is desirable. This section looks at ways in which certain restrictions can be identified in an Ada program. It then describes in detail the Ravenscar Profile which is a collection of restrictions aimed at applications that require very efficient implementations, or have high integrity requirements, or both. It provides the minimum language features necessary to program fixed priority-based applications.

Where it is necessary to produce very efficient programs, it would be useful to have run-time systems (kernels) that are tailored to the particular needs of the program actually executing. As this would be impossible to do in general, the language defines a set of restrictions that a run-time system should recognize and ‘reward’ by giving more effective support. The following example restrictions are identified by the pragma called `Restrictions`, and are checked and enforced before run-time. Note, however, that there is no requirement on the run-time to tailor itself to the restrictions specified.

- **No_Task_Hierarchy** – all (non-environment) tasks depend directly on the environment task.
- **No_Abort_Statement** – there are no abort statements.
- **No_Terminate_Alternatives** – there are no `selective_accepts` with terminate alternatives.
- **No_Task_Allocators** – there are no allocators for task types or types containing task subcomponents.
- **No_Dynamic_Priorities** – there are no semantic dependencies on the package `Dynamic_Priorities`.
- **No_Dynamic_Attachments** – there is no call to any of the operations defined in package `Interrupts`, e.g. `Attach_Handler`.
- **No_Local_Protected_Objects** – protected objects shall only be declared at library level.
- **No_Local_Timing_Events** – timing events shall only be declared at library level.

- **No.Protected.Type Allocators** – there are no allocators for protected types or types containing protected subcomponents.
- **No.Relative.Delay** – there are no relative delay statements (i.e. **delay**).
- **No.Requeue.Statements** – there are no requeue statements.
- **No.Select.Statements** – there are no select statements.
- **No.Specific.Termination.Handlers** – there are no calls to the specific handler routines in the task termination package.
- **Simple.Barriers** – the Boolean expression in an entry barrier shall either be a static Boolean expression or a Boolean component of the enclosing protected object (e.g. a simple boolean variable).

The following restrictions are also defined:

- **Max.Select.Alternatives** – specifies the maximum number of alternatives in a `selective_accept`.
- **Max.Task.Entries** – specifies the maximum number of entries per task. The maximum number of entries for each task type (including those with entry families) shall be determinable at compile-time. A value of zero indicates that no rendezvous is possible.
- **Max.Protected.Entries** – specifies the maximum number of entries per protected type. The maximum number of entries for each protected type (including those with entry families) shall be determinable at compile-time.

Subject to the implementation permissions given below, the following restrictions are checked at run-time.

- **No.Task.Termination** – all tasks are non-terminating. It is implementation defined what happens if a task terminates – but any fall-back handler must be executed as the first task terminates.
- **Max.Asynchronous.Select.Nesting** – specifies the maximum dynamic nesting level of `asynchronous_select_statements`. A value of zero prevents the use of any `asynchronous_select_statement`. If a check fails, `Storage_Error` is raised as above.
- **Max.Tasks** – specifies the maximum number of tasks, excluding the environment task, that are allowed to exist over the lifetime of a partition. A zero value prevents tasks from being created. If a check fails, `Storage_Error` is raised as above.
- **Max.Entry.Queue.Length** – This defines the maximum number of calls queued on an entry. Violation will cause `Program_Error` to be raised at point of call.

So to restrict a program to have a flat task structure and not use the **select** statement would require:

```
pragma Restrictions (No_Task_Hierarchy, No_Select_Statements);
```

In addition to these specific restrictions a program may indicate that it does not wish to make use of a predefined language package by using the restriction identifier `No_Dependence`. So for example to state that the calendar package will not be used:

```
pragma Restrictions (No_Dependence => Ada.Calendar);
```

There is increasing recognition that the software components of critical real-time applications must be provably predictable. This is particularly so for a hard real-time system, in which the failure of a component of the system to meet its timing deadline can result in an unacceptable failure of the whole system. The choice of a suitable design and development method, in conjunction with supporting tools that enable the real-time performance of a system to be analysed and simulated, can lead to a high level of confidence that the final system meets its real-time constraints.

The use of Ada has proven to be of great value within high-integrity and real-time applications, albeit via language subsets of deterministic constructs, to ensure full analysability of the code. The *Ravenscar Profile* is a subset of the tasking model, restricted to meet the real-time community requirements for determinism, schedulability analysis and memory-boundedness, as well as being suitable for mapping to a small and efficient run-time system that supports task synchronization and communication, and which could be certifiable to the highest integrity levels. The concurrency model promoted by the Ravenscar profile is consistent with the use of tools that allow the static properties of programs to be verified.

It is important to note that the Ravenscar profile is silent on the non-tasking (i.e. sequential) aspects of the language. For example, it does not dictate how exceptions should, or should not, be used. For any particular application, it is likely that constraints on the sequential part of the language will be required. These may be due to other forms of static analysis to be applied to the code, or to enable worst-case execution time information to be derived for the sequential code.

The Ravenscar profile is identified by an Ada 2005 pragma called `Profile`:

```
pragma Profile (Ravenscar);
```

A profile is a collection of restrictions and other pragmas. However, before giving the definition of Ravenscar, which concentrates on the negative aspects of the definition (i.e. what it does not contain), we first list those features that it does allow and which together provide an adequate form of tasking that can be used even for software that needs to be verified to the very highest integrity levels. Ravenscar includes:

- task types and objects, defined at the library level;
- protected types and objects, defined at the library level;
- one entry per protected object with a maximum of one task queued at any time on that entry;
- entry barriers but restricted to a single Boolean variable (or a Boolean literal);
- atomic and volatile pragmas;
- any number of **delay until** statements;

- Ceiling_Locking policy and FIFO_Within_Priorities dispatching policy;
- the E'Count attribute for protected entries except within entry barriers;
- the Ada.Task_Identification package plus task attributes T'Identity and E'Caller;
- synchronous task control;
- task type and protected type discriminants;
- the Ada.Real_Time package;
- protected procedures as statically bound interrupt handlers;
- execution time monitoring;
- library level timing events;
- fall-back task termination handlers.

Many of these features have already been considered in the book. Others are introduced later in this chapter. The profile itself is defined as follows:

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
```

```
pragma Locking_Policy (Ceiling_Locking);
```

```
pragma Detect_Blocking;
```

```
pragma Restrictions(
  No_Abort_Statements,
  No_Dynamic_Attachment,
  No_Dynamic_Priorities,
  No_Implicit_Heap_Allocations,
  No_Local_Protected_Objects,
  No_Local_Timing_Events,
  No_Protected_Type_Allocators,
  No_Relative_Delay,
  No_Requeue_Statements,
  No_Select_Statements,
  No_Specific_Termination_Handlers,
  No_Task_Allocators,
  No_Task_Hierarchy,
  No_Task_Termination,
  Simple_Barriers,
  Max_Entry_Queue_Length => 1,
  Max_Protected_Entries => 1,
  Max_Task_Entries => 0,
  No_Dependence => Ada.Asynchronous_Task_Control,
  No_Dependence => Ada.Calendar,
  No_Dependence => Ada.Execution_Time.Group_Budget,
  No_Dependence => Ada.Execution_Time.Timers,
  No_Dependence => Ada.Task_Attributes);
```

Most of these restrictions should be easy to understand. They come from a need to have a static program with a fixed set of tasks and protected objects. As communication

via protected objects is amenable to analysis, the rendezvous and the consequential use of select statements is prohibited. Other restrictions are motivated by the need to have a small and efficient run-time support system – hence the use of abort statements and complex barriers is disallowed. The protocol required for implementing general protected objects is significantly simplified by the restriction of one entry per object and one queued task per entry. Non-static features such as the dynamic attachment of interrupt handlers and task-specific termination handlers are excluded as is the use of package `Calendar`; the real-time clock package is the preferred time base to use. Note that two of the restrictions, `No_Task_Termination` and `Max_Entry_Queue_Length => 1`, result in run-time checks (see the definitions of these restrictions).

An important point to emphasis about Ravenscar is that it is not a substitute for the full tasking model. It has significantly less expressive power, and many applications needs cannot be programmed with this profile.

To complete the definition, the pragma `Detect_Blocking` needs to be defined. Within a protected operation it is illegal to block, for example to call an external procedure that has a delay statement within it. Although illegal it is not possible for the compiler to check that this cannot occur, so it becomes a run-time issue. The program is said to experience a **bounded error** which can result in a number of possible behaviours, one of which is for the exception `Program_Error` to be raised. What the pragma `Detect_Blocking` does is ensure that the error condition is recognized and `Program_Error` raised. Although this checking will add to the overheads, i.e. detract from real-time performance, it is deemed necessary for high-integrity applications.

12.5 Dynamic priorities and other Ada facilities

The above discussions have assumed that the base priority of a task remains constant during the entire existence of the task. This is an adequate model for many scheduling approaches. There are, however, situations in which it is necessary to alter base priorities. For example:

- to implement mode changes;
- to implement application-specific scheduling schemes.

In the first example, base priority changes are infrequent and correspond to changes in the relative temporal properties of the tasks after a mode change (for example, a task running more frequently in the new mode). The second use of dynamic priorities allows programmers to construct their own schedulers. An example of this is the programming of execution-time servers that will be illustrated in the Section 13.6.3.

To support dynamic priorities, the language provides a library package – see Program 12.1. The function `Get_Priority` returns the current base priority of the task; this can be changed by the use of `Set_Priority`. A change of base priority takes effect immediately the task is outside a protected object.

Ada 2005 also allows the ceiling priority of a protected object to be changed at run-time. For any protected object, `P`, the attribute `P'Priority` represents a component of `P` of type `System.Any_Priority`. References to the `Priority` attribute can

Program 12.1 Dynamic priority package.

```

with Ada.Task_Identification;
with System;
package Ada.Dynamic_Priorities is
  procedure Set_Priority(Priority : System.Any_Priority;
                        T : Task_Identification.Task_Id :=
                        Task_Identification.Current_Task);
    -- raises Program_Error if T is the Null_Task_Id
    -- has no effect if the task has terminated

  function Get_Priority(T : Task_Identification.Task_Id :=
                        Task_Identification.Current_Task)
    return System.Any_Priority;
    -- raises Tasking_Error if the task has terminated
    -- or Program_Error if T is the Null_Task_Id
private
  -- not specified by the language
end Ada.Dynamic_Priorities;

```

only occur from within the body of the associated protected body. Such references can be read or write. If the locking policy `Ceiling_Locking` is in effect then a change to the `Priority` attribute results in the ceiling value changing to this new value – but only at the end of the protected action that resulted in the change.

As well as changing the priority of a task, which obviously affects the likelihood of it being scheduled, Ada also allows a task to suspend itself and to suspend other tasks. These two facilities are called **synchronous task control** (see Program 5.1) and **asynchronous task control**. The latter is supported by the following package:

```

with Ada.Task_Identification;
package Ada.Asynchronous_Task_Control is
  procedure Hold(T : Task_Identification.Task_Id);
  procedure Continue(T : Task_Identification.Task_Id);
  function Is_Held(T : Task_Identification.Task_Id)
    return Boolean;
end Ada.Asynchronous_Task_Control;

```

For each processor, there is a conceptual idle task which can always run (but has a priority below any application task). A call of `Hold` lowers the base priority of the designated task to below that of the idle task. It is said to be *held*. If the designated task is not executing with an inherited priority, it will be suspended immediately. A call of `Continue` restores the task's priority. This facility will be used in the next chapter to program a type of execution-time server.

Ada also provides other facilities which are useful for programming a wide variety of real-time systems that are scheduled using fixed priorities, for example, prioritized entry queues, task attributes, and so on. There are also a number of other dispatching policies. One for EDF scheduling is defined later in this chapter (Section 12.8). Another defines non-preemptive scheduling: `Non_Preemptive_FIFO_Within_Priorities`.

This has similar properties to the preemptive version other than the obvious requirement for a task to continue executing even when a higher-priority task becomes runnable. Interrupts can still occur, but task switching is postponed until the task itself executes a blocking operation such as a delay statement. Note the execution of 'delay 0.0' is sufficient to end a period of non-preemptive execution. A further dispatching policy defined in the real-time annex is Round-Robin scheduling. This is the usual algorithm where tasks of the same priority are allocated a quantum of execution time before they are suspended and return to the back of the queue for that priority level. To ensure that the integrity of protected objects is not compromised by this policy then suspension is itself suspended whilst a task executes within such objects.

As well as supporting a number of distinct dispatching policies Ada also allows mixed systems to be specified. This is described later in this chapter once EDF scheduling has been defined (see Section 12.9).

The reader is referred to the Systems Programming and Real-Time Annexes of the Ada Reference Manual and the Further Reading section of this chapter for details on all Ada facilities in the area of real-time programming.

12.6 C/Real-Time POSIX and fixed-priority scheduling

C/Real-Time POSIX consists of a variety of related standards. There is the base standard, the real-time extensions, the threads extensions and so on. If implemented in a single system, it would contain a huge amount of software. To help produce more compact versions of operating systems which conform to the C/Real-Time POSIX specifications, a set of application environment **profiles** have been developed, the idea being that implementors can support one or more profiles. For real-time systems, four profiles have been defined in C/Real-Time POSIX:

- **PSE51** – minimal real-time system profile – intended for small single/multi-processor embedded systems controlling one or more external devices; no operator interaction is required and there is no file system. The main services are threads, fixed-priority scheduling, mutexes with priority inheritance, condition variables, semaphores, simple device I/O and signals. It is analogous to the Ravenscar Profile.
- **PSE52** – real-time controller system profile – an extended PSE51 for potentially multiple processors with a file system interface, message queues and tracing facilities.
- **PSE53** – dedicated real-time system profile – an extension of PSE52 for single or multiprocessor systems; includes multiple multithreaded processes and asynchronous I/O.
- **PSE54** – multipurpose real-time system profile – capable of running a mix of real-time and non real-time processes executing on single/multiprocessor systems with memory management units, mass storage devices, networks and so on.

In general, a C/Real-Time POSIX system is also free not to support any of the optional units of functionality it chooses, and so much finer control over the supported

functionality is possible. All of the real-time and the thread extensions are optional. However, conforming to one of the profiles means that all the required units of functionality must be supported.

To support priority-based scheduling, C/Real-Time POSIX has options to support priority inheritance and ceiling protocols. Priorities may be set dynamically. Within the priority-based facilities, there are four policies.

- **FIFO** – a process/thread runs until it completes or it is blocked; if a process/thread is preempted by a higher-priority process/thread then it is placed at the head of the run queue for its priority.
- **Round-Robin** – a process/thread runs until it completes or it is blocked or its time quantum has expired; if a process/thread is preempted by a higher-priority process then it is placed at the head of the run queue for its priority; however, if its quantum expires it is placed at the back.
- **Sporadic Server** – a process/thread runs as a sporadic server (see Section 11.6.2).
- **OTHER** – an implementation-defined policy (which must be documented).

For each scheduling policy, there is a minimum range of priorities that must be supported; for FIFO and round-robin, this must be at least 32. The scheduling policy can be set on a per process and a per thread basis.

Threads may be created with a ‘system contention’ option, in which case they compete with other system threads according to their policy and priority. Alternatively, threads can be created with a ‘process contention’ option; in this case they must compete with other threads (created with a process contention) in the parent process. It is unspecified how such threads are scheduled relative to threads in other processes or to threads with global contention. A specific implementation must decide whether to support ‘system contention’ or ‘process contention’ or both.

Programs 12.2 and 12.3 illustrate the C interface to the Real-Time POSIX scheduling facilities. The functions are divided into those which manipulate a process’s scheduling policy and parameters, and those that manipulate a thread’s scheduling policy and parameters. If a thread modifies the policy and parameters of its owning process, the effect on the thread will depend upon its contention scope (or level). If it is contending at a system level, the change will not affect the thread. If, however, it is contending at the process level, there will be an impact on the thread.

In order to prevent priority inversion, C/Real-Time POSIX allows inheritance protocols to be associated with mutex variables. As well as catering for basic priority inheritance, the immediate ceiling priority protocol (called the priority protect protocol by C/Real-Time POSIX) is also supported.

C/Real-Time POSIX provides other facilities that are useful for real-time systems. For example, it allows:

- message queues to be priority ordered;
- functions for dynamically getting and setting a thread’s priority;
- threads to indicate whether their attributes should be inherited by any child thread they create.

Program 12.2 The C/Real-Time POSIX interface to the Real-Time POSIX scheduling facilities.

```

#define SCHED_FIFO ...      /* preemptive priority scheduling */
#define SCHED_RR ...       /* preemptive priority with quantum */
#define SCHED_SPORADIC ... /* sporadic server */
#define SCHED_OTHER ...    /* implementation-defined scheduler */
#define PTHREAD_SCOPE_SYSTEM ... /* system-wide contention */
#define PTHREAD_SCOPE_PROCESS ... /* local contention */
#define PTHREAD_PRIO_NONE ... /* no priority inheritance */
#define PTHREAD_PRIO_INHERIT ... /* basic priority inheritance */
#define PTHREAD_PRIO_PROTECT ... /* ICPP */

typedef ... pid_t;

struct sched_param {
    ...
    int sched_priority; /* used for SCHED_FIFO and SCHED_RR */
    ... };

int sched_setparam(pid_t pid, const struct sched_param *param);
int sched_getparam(pid_t pid, struct sched_param *param);
/* set/get the scheduling parameters of process pid */

int sched_setscheduler(pid_t pid, int policy,
                      const struct sched_param *param);
int sched_getscheduler(pid_t pid);
/* set/get the scheduling policy of process pid */

int sched_yield(void);
/* causes the current thread/process to be placed at the back */
/* of the run queue */

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
/* returns the max/minimum priority for the policy specified */

int sched_rr_get_interval(pid_t pid, struct timespec *t);
/* if pid != 0, the time quantum for the calling process/thread is
   set in the structure referenced by t
   if pid = 0, the calling process/threads time quantum is set
   in the structure pointed to by t
   */

```

12.7 Real-Time Java and fixed-priority scheduling

Real-Time Java has the notion of a schedulable object. This is any object that supports the `Schedulable` interface given in Program 12.4. The classes `RealtimeThread`, `NoHeapRealTimeThread` and `AsyncEventHandler` all support this interface. Objects of these classes all have scheduling parameters (see Program 12.5). Real-Time Java implementations are required to support at least 28 real-time priority levels. As with

Program 12.3 The C/Real-Time POSIX interface to the Real-Time POSIX scheduling facilities continued.

```

int pthread_attr_setscope(pthread_attr_t *attr,
                          int contentscope);
int pthread_attr_getscope(const pthread_attr_t *attr,
                          int *contentscope);
/* set/get the contention scope attribute for a */
/* thread attribute object */

int pthread_attr_setschedpolicy(pthread_attr_t *attr,
                                int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr,
                                int *policy);
/* set/get the scheduling policy attribute for a */
/* thread attribute object */

int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *param);
int pthread_attr_getschedparam(const pthread_attr_t *attr,
                               struct sched_param *param);
/* set/get the scheduling policy attribute for a */
/* thread attribute object */

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
                                  int protocol);
int pthread_mutexattr_getprotocol(pthread_mutexattr_t *attr,
                                  int *protocol);
/* set/get the priority inheritance protocol */

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
                                     int prioceiling);
int pthread_mutexattr_getprioceiling(pthread_mutexattr_t *attr,
                                     int *prioceiling);
/* set/get the priority ceiling */

/* All the above integer functions return 0 if successful */

```

Ada and C/Real-Time POSIX, the larger the integer value, the higher the priority (and, therefore, the greater the execution eligibility). Non-real-time threads are given priority levels below the minimum real-time priority. Note, scheduling parameters are bound to threads at thread creation time (see Program 10.1). If the parameter objects are changed, they have an immediate impact on the associated threads.

In common with Ada and C/Real-Time POSIX, Real-Time Java supports a preemptive priority-based dispatching policy. However, unlike Ada and C/Real-Time POSIX, Real-Time Java does not require (but does recommend) a preempted thread to be placed at the head of the run queue associated with its priority level. Real-Time Java also supports a high-level scheduler whose goals are to decide whether to admit new schedulable objects according to the resources available and a feasibility algorithm. Hence, while Ada and C/Real-Time POSIX focus on static offline schedulability analysis, Real-Time Java addresses more dynamic systems with the potential for online analysis.

Program 12.4 An extract of the Real-Time Java Schedulable interface.

```

public interface Schedulable extends java.lang.Runnable {
    ...
    public void addToFeasibility();
    public void removeFromFeasibility();

    public MemoryParameters getMemoryParameters();
    public void setMemoryParameters(MemoryParameters memory);

    public ReleaseParameters getReleaseParameters();
    public void setReleaseParameters(ReleaseParameters release);

    public SchedulingParameters getSchedulingParameters();
    public void setSchedulingParameters(
        SchedulingParameters scheduling);

    public Scheduler getScheduler();
    public void setScheduler(Scheduler scheduler);
}

```

Program 12.5 The Real-Time Java SchedulingParameters class and its subclasses.

```

public abstract class SchedulingParameters {
    public SchedulingParameters();
}

public class PriorityParameters extends SchedulingParameters {
    public PriorityParameters(int priority);

    public int getPriority();
    public void setPriority(int priority) throws
        IllegalArgumentException;
    ...
}

public class ImportanceParameters extends PriorityParameters {
    public ImportanceParameters(int priority, int importance);
    public int getImportance();
    public void setImportance(int importance);
    ...
}

```

The Scheduler abstract class is given in Program 12.6. The `isFeasible` method considers only the set of schedulable objects that have been added to its feasibility list (via the `addToFeasibility` and `removeFromFeasibility` methods). The method `changeIfFeasible` checks to see if its set of objects is still feasible if the given object has its release and memory parameters changed. If it is, the parameters are changed. Static methods allow the default scheduler to be queried or set.

Program 12.6 The Real-Time Java Scheduler class.

```

public abstract class Scheduler {
    protected Scheduler();

    protected abstract void addToFeasibility(
        Schedulable schedulable);
    protected abstract void removeFromFeasibility(
        Schedulable schedulable);

    public abstract boolean isFeasible();
    // checks the current set of schedulable objects

    public boolean changeIfFeasible(Schedulable schedulable,
        ReleaseParameters release,
        MemoryParameters memory);

    public static Scheduler getDefaultScheduler();
    public static void setDefaultScheduler(Scheduler scheduler);

    public abstract java.lang.String getPolicyName();
}

```

One defined subclass of the Scheduler class is the PriorityScheduler class (defined in Program 12.7), which implements standard preemptive priority-based scheduling.

Again, it should be stressed that Real-Time Java does not require an implementation to provide an online feasibility algorithm. The default algorithm assumes an adequately fast computer. Consequently, it returns `true` – if the application contains only periodic and sporadic schedulable objects; `false` otherwise (that is, if aperiodic schedulable objects are present).

Real-Time Java allows priority inheritance algorithms to be used when accessing synchronized classes. To achieve this, three classes are defined (see Program 12.8). Consider the following class:

```

public class SynchronizeClass {
    public void Method1() { ... };
    public void Method2() { ... };
}

```

An instance of this class can have its control protocol set to immediate priority ceiling inheritance (called priority ceiling emulation by Real-Time Java) with a priority of 10 by the following code:

```

SynchronizeClass SC = new SynchronizeClass();
PriorityCeilingEmulation PCI = new PriorityCeilingEmulation(10);
...
MonitorControl.setMonitorControl(SC, PCI);

```

It is important to note that all queues in Real-Time Java are priority ordered.

Program 12.7 The Real-Time Java PriorityScheduler class.

```

class PriorityScheduler extends Scheduler {
    public PriorityScheduler()

    protected void addToFeasibility(Schedulable s);
    protected void removeFromFeasibility(Schedulable s);

    public boolean isFeasible();
    // checks the current set of schedulable objects

    public boolean changeIfFeasible(Schedulable schedulable,
                                     ReleaseParameters release, MemoryParameters memory);

    protected void addToFeasibility(Schedulable s);
    protected void removeFromFeasibility(Schedulable s);

    public void fireSchedulable(Schedulable schedulable);

    public int getMaxPriority();
    public int getMinPriority();
    public int getNormPriority();
    public java.lang.String getPolicyName();

    public static PriorityScheduler instance();

    ...
}

```

Like Ada and C/Real-Time POSIX, Real-Time Java has very comprehensive real-time support facilities. Similarly, profiles have been defined. Three such profiles are those developed by the JSR 302 Expert Group.² Their goal is to define subsets of Real-Time Java for use in safety-critical systems. This requires a much tighter and smaller set of Java virtual machines and libraries with much more precise performance requirements. The following three profiles have been defined – here the focus is on the scheduling models rather than the memory management models.

- **Level 0** – a Level 0 application's programming model is similar to a cyclic executive model as defined in Section 12.1. A Level 0 application's schedulable objects shall consist only of a set of periodic asynchronous event handlers. Each handler has a period, priority and start time relative to the beginning of a major cycle. A schedule of all handlers is constructed by either the application designer or by an offline tool provided with the implementation. All handlers in a Level 0 implementation execute using a single server thread.
- **Level 1** – a Level 1 application uses a fixed-priority programming model with a set of concurrent computations, each with a priority, running under the control of

²See <http://jcp.org/en/jsr/detail?id=302>.

Program 12.8 Real-Time Java classes supporting priority inheritance.

```

public abstract class MonitorControl {
    public MonitorControl();

    public static void setMonitorControl(MonitorControl policy);
    // set the default

    public static void setMonitorControl(java.lang.Object monitor,
                                         MonitorControl policy);
    // sets an individual objects policy
}

public class PriorityCeilingEmulation extends MonitorControl {
    public PriorityCeilingEmulation(int ceiling);

    public int getDefaultCeiling();
    // get ceiling for this object
}

public class PriorityInheritance extends MonitorControl {
    public PriorityInheritance();

    public static PriorityInheritance instance();
}

```

a fixed-priority preemptive scheduler. The computations are performed in a set of periodic and sporadic event handlers. Only mutual exclusion is supported; there is no support for condition synchronization (that is, the use of the `wait` and `notify` methods is prohibited).

- **Level 2** – computation at Level 2 is performed in a set of asynchronous event handlers and/or `NoHeapRealtimethreads`. A Level 2 application may use the `wait` and `notify` methods.

As indicated above, the profiles also address the use of dynamic memory allocation. The Ada and C/Real-Time POSIX profiles are silent on this issue.

12.8 Programming EDF systems

To support the programming of applications that wish to make use of EDF scheduling, three language features are required:

- a formal representation of the deadline of a task;
- use of deadlines to control dispatching;
- a means of sharing data between tasks that is compatible with EDF.

Program 12.9 The EDF dispatching package.

```

with Ada.Real_Time; with Ada.Task_Identification;

package Ada.Dispatching.EDF is
  subtype Deadline is Ada.Real_Time.Time;
  Default_Deadline : constant Deadline :=
    Ada.Real_Time.Time_Last;
  procedure Set_Deadline(D : in Deadline;
    T : in Ada.Task_Identification.Task_ID :=
      Ada.Task_Identification.Current_Task);
  procedure Delay_Until_And_Set_Deadline(
    Delay_Until_Time : in Ada.Real_Time.Time;
    TS : in Ada.Real_Time.Time_Span);
  function Get_Deadline(T : in Ada.Task_Identification.Task_ID :=
    Ada.Task_Identification.Current_Task) return Deadline;
end Ada.Dispatching.EDF;

```

Of all the mainstream engineering languages discussed in this book only Ada gives direct support to EDF scheduling. Both C/Real-Time POSIX and Real-Time Java allow implementations to support EDF scheduling, but not in a portable way. This section will therefore only discuss Ada, and will cover the three feature just identified. Note Ada's support for deadline-based scheduling does not extend to any notion of deadline inheritance.

12.8.1 Representing deadlines in Ada

A task's deadline can be set using the facilities defined in Program 12.9.

The identifier `Deadline` is explicitly introduced even though it is a direct subtype of the time type from `Ada.Real_Time`.

The `Set` and `Get` subprograms have obvious utility. A call of `Delay_Until_And_Set_Deadline` delays the calling task until time `Delay_Until_Time`. When the task becomes runnable again it will have deadline `Delay_Until_Time + TS`. The inclusion of this procedure reflects a common task structure for periodic activity. Consider the example of a periodic task; now assume it has a deadline at the end of its execution (i.e. every time it executes it should finish before its next release):

```

with Ada.Real_Time; use Ada.Real_Time;
with Ada.Dispatching.EDF; use Ada.Dispatching.EDF;
...

task Periodic_Task;

task body Periodic_Task is
  Interval : Time_Span := Milliseconds(30);
  -- define the period of the task, 30ms in this example
  Next : Time;
begin
  Next := Clock; -- start time
  Set_Deadline(Clock+Interval);

```



```

loop
  -- undertake the work of the task
  Next := Next + Interval;
  Delay_Until_And_Set_Deadline(Next, Interval);
end loop;
end Periodic_Task;

```

If, rather than using just this one procedure call, the task had a set deadline call and a delay until statement then this would most likely result in an extra unwanted task switch (first the deadline is extended and hence a more urgent task will preempt, later the task will execute again just to put itself on the delay queue).

With EDF, all dispatching decisions are based on deadlines, and hence it is necessary for a task to always have a deadline.³ However, a task must progress though activation before it can get to a position to call `Set_Deadline` and hence a default deadline value is given to all tasks (`Default_Deadline` defined in `Ada.Dispatching.EDF`). However, this default value is well into the future and hence activation will take place with very low urgency (all other task executions will occur before this task's activation). If more urgency is required, the following language-defined pragma is available for inclusion in a task's specification (only):

```
pragma Relative_Deadline(Relative_Deadline_Expression);
```

where the type of the parameter is `Ada.Real_Time.Time_Span`. The initial absolute deadline of a task containing this pragma is the value of `Ada.Real_Time.Clock + Relative_Deadline_Expression`, the call of the clock being made between task creation and the start of its activation.

So the example above should use this pragma rather than include the first call of `Set_Deadline`:

```

task Periodic_Task is
  pragma Relative_Deadline(Milliseconds(30));
end Periodic_Task;

```

A final point to note with the deadline assignment routine concerns when it will take effect. The usual result of a call to `Set_Deadline` is for the associated deadline to be increased into the future and that a task switch is then likely (if EDF dispatching is in force). As this would not be appropriate if the task is currently executing within a protected object the setting of a task's deadline to the new value takes place as soon as is practical but not while the task is performing a protected action. This is similar to the rule that applies to changes to the base priority of a task using the dynamic priority facility.

Once a deadline is set then it becomes possible to check at run-time that the program's execution does indeed meet its deadlines. This is easily accommodated by the asynchronous transfer of control (select-then-abort) feature:

```

loop
  select
    delay until Ada.Dispatching.EDF.Get_Deadline;

```

³This is one of the criticisms of EDF – even tasks that have no actual deadline must be given an artificial one so that they will be scheduled.

```

    -- action to be take when deadline missed
  then abort
    -- code
  end select;
end loop;

```

A fuller description of the options available to the programmer when a deadline is missed is given in Chapter 13.

12.8.2 Dispatching

To request EDF dispatching, the following use of the dispatching policy pragma is supported:

```
pragma Task_Dispatching_Policy(EDF_Across_Priorities);
```

The main result of employing this policy is that the ready queues are now ordered according to deadline (not FIFO). Each ready queue is still associated with a single priority, but at the head of any ready queue is the runnable task with the earliest deadline. Whenever a task is added to a ready queue for its priority it is placed in the position dictated by its current absolute deadline.

The active priority of a task, under EDF dispatching, is no longer directly linked to the base priority of the task. The rules for computing the active priority of a task are somewhat complex and are covered in the next section – they are derived from consideration of each task's use of protected objects. For a simple program with no protected objects the following straightforward rules apply:

- any priorities set by the tasks are ignored;
- all tasks are always placed on the ready queue for priority value `System.Priority'First`.⁴

Hence only one ready queue is used (and that queue is ordered by deadline).

Of course real programs require task interactions, and for Ada real-time programs this usually means the use of protected objects. To complete the definition of the EDF dispatching policy the rules for using protected objects must be considered. This is outlined below.

12.8.3 EDF and Baker's algorithm

In Section 11.11.4, Baker's algorithm was introduced as a means of controlling access to shared objects when EDF scheduling was in force. Ada supports Baker's approach by:

- using deadline to represent urgency;
- using base priority to represent each task's preemption level;

⁴If EDF dispatching is defined just for a range of priorities then this priority value is the initial (lowest) value in that range – see Section 12.9.

- using ceiling priorities to represent preemption levels for protected objects;
- using standard `Ceiling_Locking` for access to protected objects.

So tasks have a base priority, but this is not used directly to control dispatching. EDF controls dispatching; the base priority only defines a preemption level.

Baker's algorithm states that a newly released task, T1 say, preempts the currently running task, T2, if and only if:

- the deadline of T1 is earlier than the deadline of T2; and
- the preemption level of T1 is higher than the preemption of any locked protected object (i.e. protected objects that are currently in use by any other task).

To keep track of all locked protected objects is an implementation overhead and hence the rules defined for Ada have the following form. Remember that if `EDF_Across_Priorities` is defined then all ready queues within the range `Priority'First .. Priority'Last` are ordered by deadline. Now rather than always place tasks in the queue for `Priority'First` the following rules apply:

- Whenever a task T is added to a ready queue, other than when it is preempted, it is placed on the ready queue with the highest priority R, if one exists, such that:
 - another task, S, is executing within a protected object with ceiling priority R; and
 - task T has an earlier deadline than task S; and
 - the base priority (preemption level) of task T is greater than R.

If no such ready queue exists the task is added to the ready queue for `Priority'First`.

- When a task is chosen for execution it runs with the active priority of the ready queue from which the task was taken. If it inherits a higher active priority it will return to its original active priority when it no longer inherits the higher level.

It follows that if no protected objects are in use at the time of the release of T then T will be placed in ready queue at level `Priority'First` at the position dictated by its deadline.

A potential task switch occurs for the currently running task T whenever:

- a change to the deadline of T takes effect; or
- a decrease to the deadline of any task on a ready queue for that processor takes effect and the new deadline is earlier than that of the running task; or
- there is a non-empty ready queue for that processor with a higher priority than the priority of the running task.

So dispatching is preemptive, but it may not be clear that the above rules implement Baker's algorithm. Consider four scenarios. Remember in all of these behaviours, the running task is always returned to its ready queue whenever a task arrives. A task (possibly the same task) is then chosen to become the running task following the rules defined above.

Task	Relative deadline <i>D</i>	Preemption level <i>L</i>	Uses resources	Arrives at time	Absolute deadline <i>A</i>
T1	100	1	R1,R3	0	100
T2	80	2	R2,R3	2	82
T3	60	3	R2	4	64
T4	40	4	R1	8	48

Table 12.1 A task set (time attributes in milliseconds).

Protected object	Ceiling value
R1	4
R2	3
R3	2

Table 12.2 Ceiling values.

The system contains four tasks; T1, T2, T3 and T4; and three resources that are implemented as protected objects: R1, R2 and R3. Table 12.1 defines the parameters of these entities.

Consider just a single invocation of each task. The arrival times have been chosen so that the tasks arrive in order of lowest preemption level task first, etc. Assume all computation times are sufficient to cause the executions to overlap.

The resources are all used by more than one task, but only one at a time and hence the ceiling values of the resources are straightforward to calculate. For R1, it is used by T1 and T4; hence the ceiling preemption level is 4. For R2, it is used by T2 and T3; hence the ceiling value is 3. Finally, for R3, it is used by T1 and T2; the ceiling equals 2 (see Table 12.2).

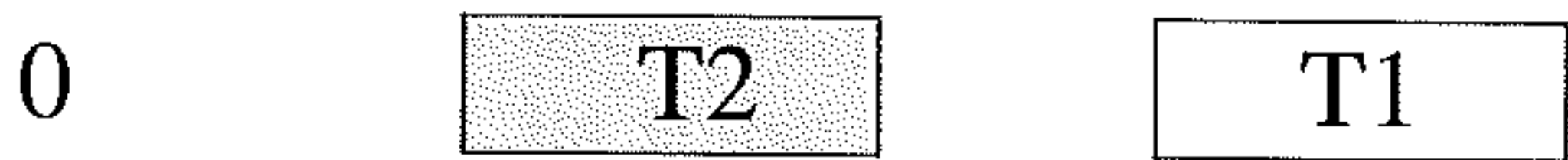
To implement this set of tasks and resources will require ready queues at level 0 (value of Priority' First in this example) and values 2, 3 and 4.

Scenario 1

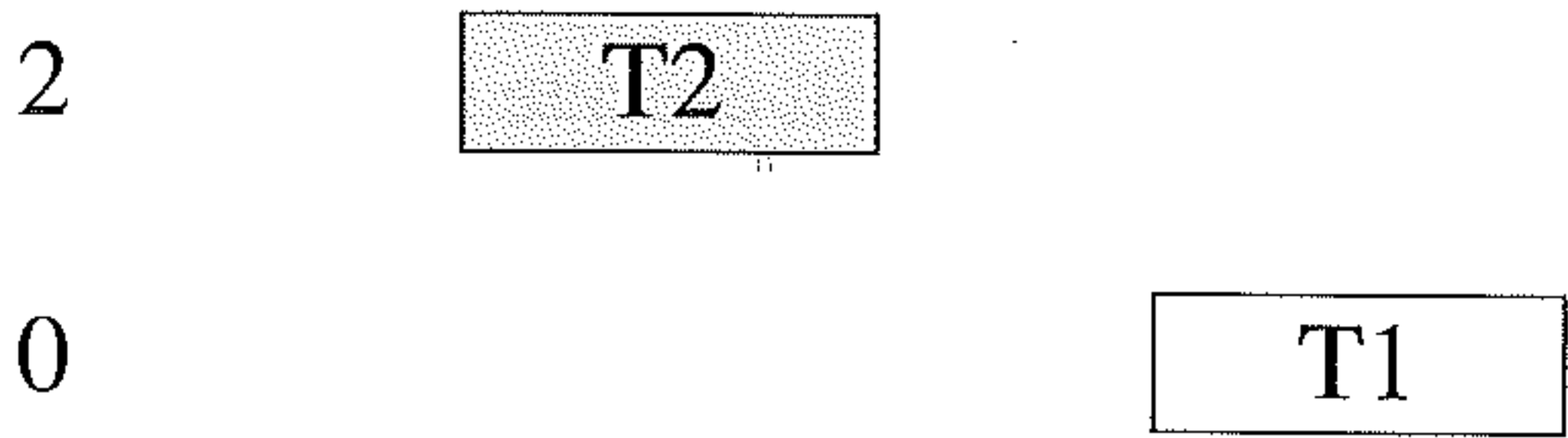
At time 0, T1 arrives. All ready queues are empty and all resources are free so T1 is placed in queue 0. It becomes the running/executing task. This is illustrated in the following where 'Level' is the priority level, 'executing' is the name of the task that is currently executing, and 'Ready Queue' shows the other executable tasks in the system. Again remember that the executing task is not on a ready queue, but is returned to a ready queue before any dispatching decision is taken.

Level	Executing	Ready Queue
0	<div>T1</div>	

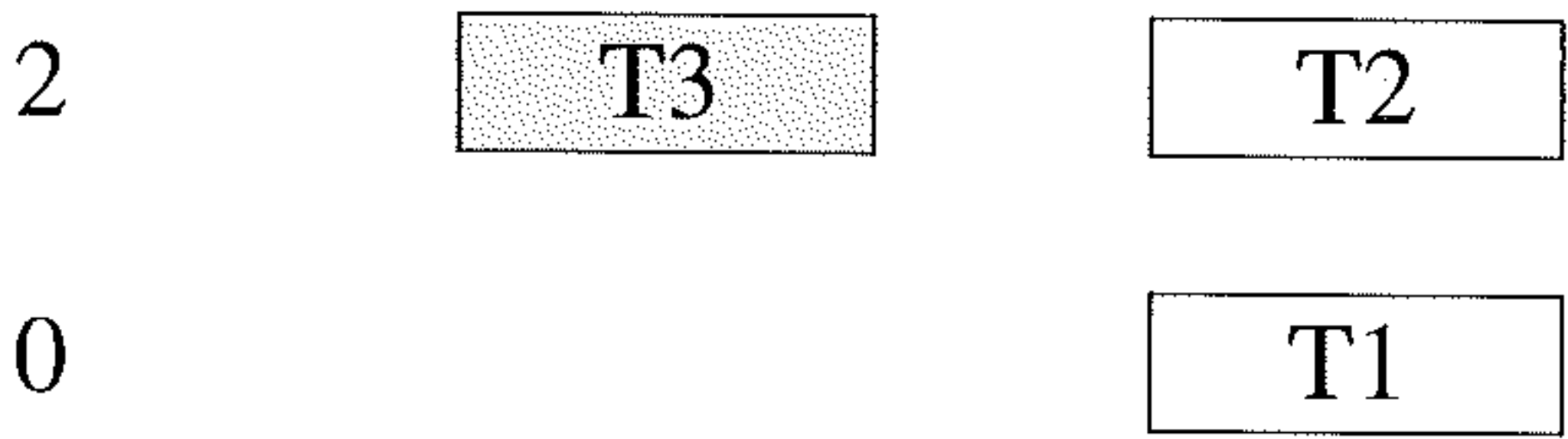
At time 2, T2 arrives and is added to ready queue 0 in front of T1 as it has a shorter absolute deadline. Now T2 is chosen for execution.



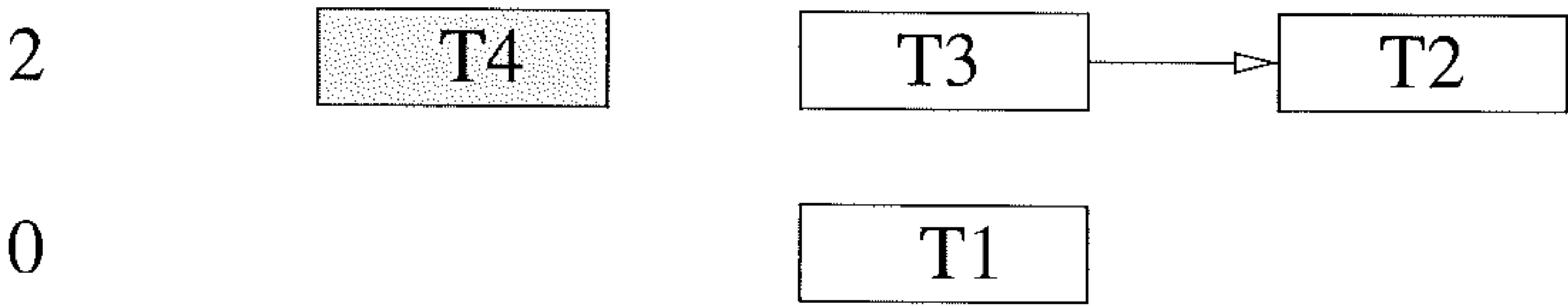
Assume at time 3, T2 calls R3. Its active priority will rise to 2.



At time 4, T3 arrives. Task T2 is joined by T3 on queue 2, as T3 has an earlier deadline and a higher preemption level; T3 is at the head of this queue and becomes the running task.



At time 8, T4 arrives. Tasks T3 and T2 are now joined by T4 as it has a deadline earlier than T3 and a higher preemption level (than 2). Task T4 now becomes the running task, and will execute until it completes; any calls it makes on resource R1 will be allowed immediately as this resource is free.

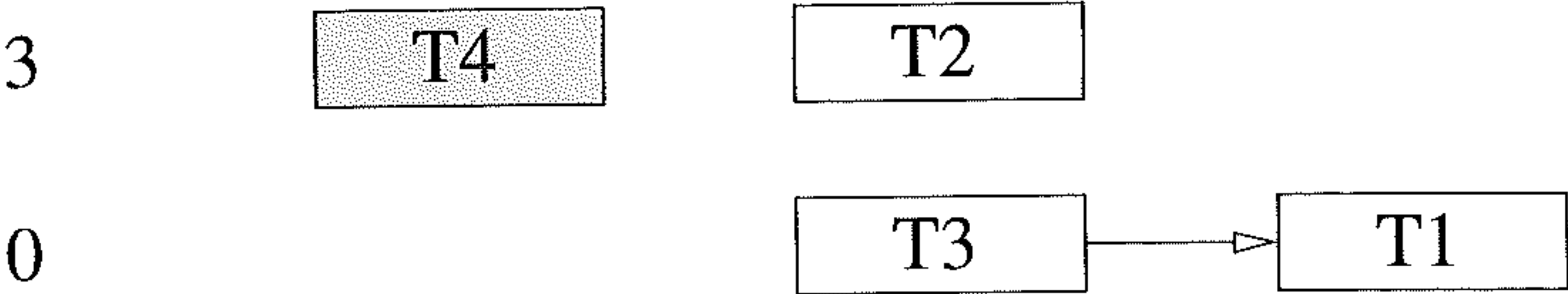


At some time later, T4 will complete then T3 will execute (at priority 2, or 4 if it locks R2) then when it completes, T2 will execute (also at priority 2) until it releases resource R3, at which point its priority will drop to 0 but it will continue to execute. Eventually when T2 completes, T1 will resume (initially at priority 0 – but this will rise if it accesses either of the resources it uses).

Scenario 2

Here we make the simple change that, at time 3, T2 calls R2 instead of R3. Its active priority will rise to 3. Now when T3 arrives at time 4, it will not have a high enough preemption level to join ready queue 3 and will be placed on the lowest queue at level 0 (but ahead of T1). Task T2 continues to execute.

At time 8, T4 arrives. It passes both elements of the test and is placed on the queue at level 3 ahead of T2 and therefore preempts it.

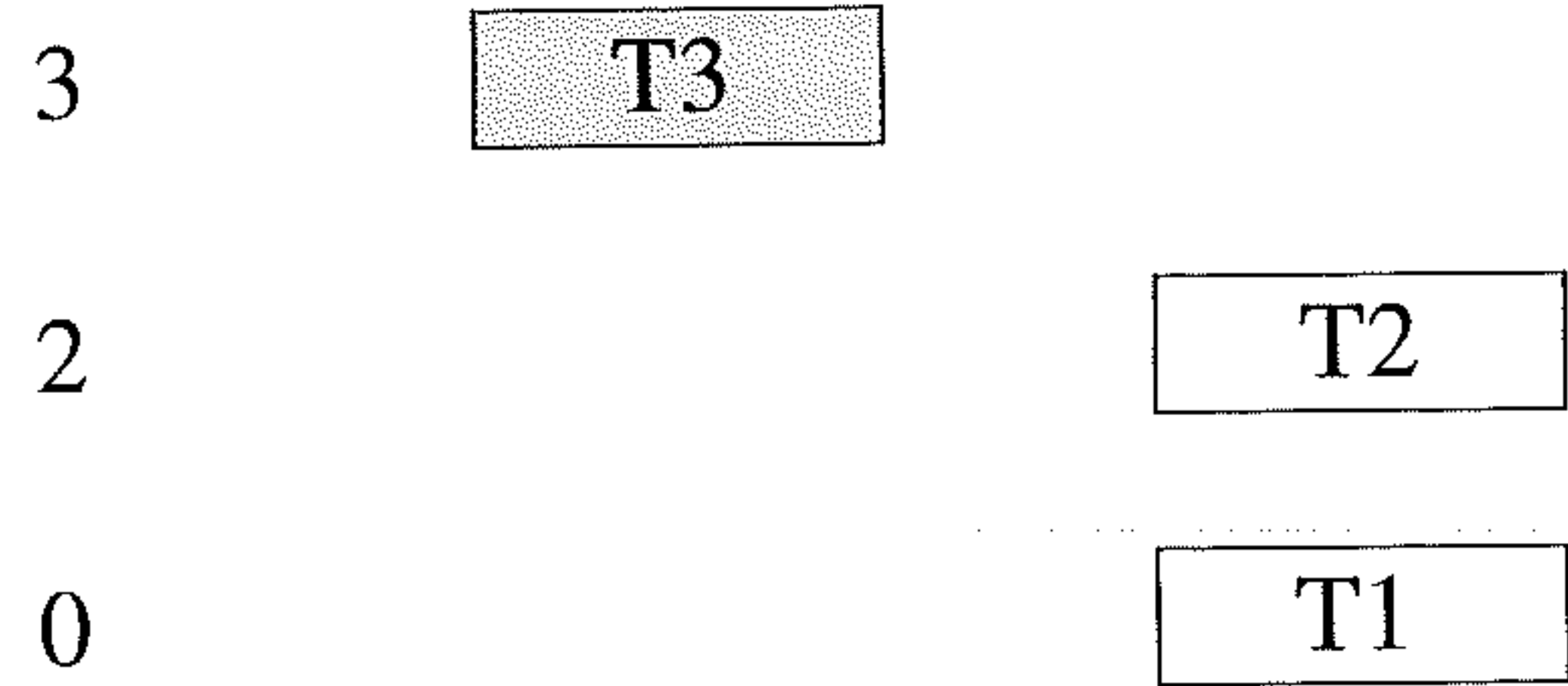


At some time later, T4 will complete then T2 will execute (at priority 3) until it releases resource R2, at which point its priority will drop to 0. Now T3 will preempt and becomes the running task.

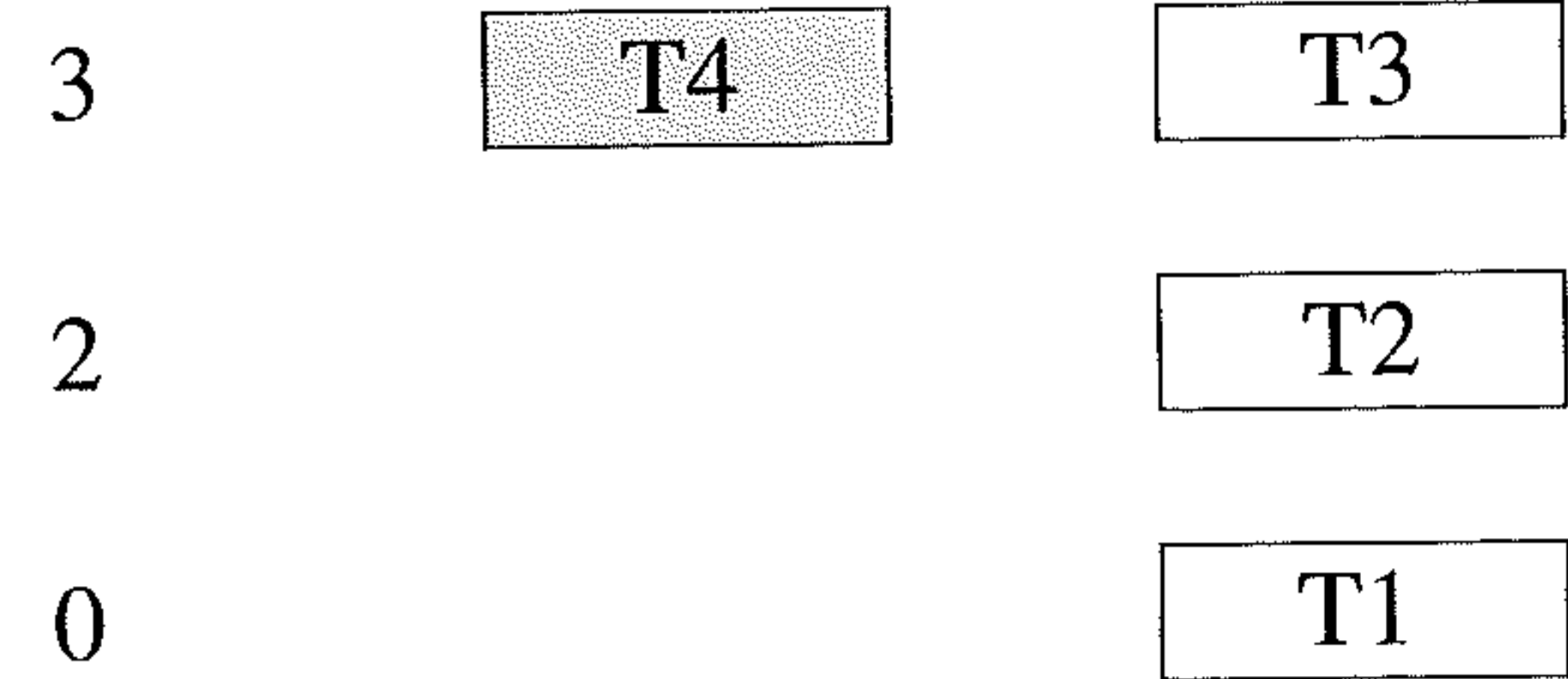
Scenario 3

For another example, return to the first scenario but assume T3 makes use of resource R2 before T4 arrives.

- At time 0, T1 arrives. All ready queues are empty and all resources are free so T1 is placed in queue 0. It becomes the running task.
- At time 2, T2 arrives and is added to ready queue 0 in front of T1.
- Assume at time 3, T2 calls R3. Its active priority will rise to 2.
- At time 4, T3 arrives and becomes the running task at priority level 2.
- At time 5, T3 calls R2 (note all resource requests are always to resources that are currently free) and thus its active priority rises to 3.

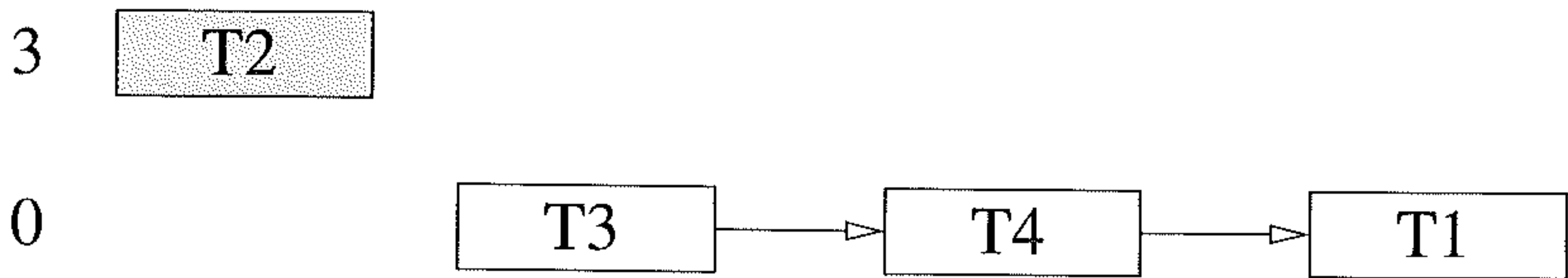


- At time 8, T4 arrives. There is now one task on queue 0; one on queue 2 (T2 holding resource R3) and one task on queue 3 (T3 holding R2). The highest ready queue that the dispatch rules determines is that at level 3, and hence T4 joins T3 on this queue – but at the head and hence becomes the running task.



Scenario 4

Now consider a simple change to the parameters – let the relative deadline of T4 be 58 so that its absolute deadline is 66 which is later than T3’s absolute deadline. At time 3, T2 calls R2. Its active priority will rise to 3. Now when T3 arrives at time 4, it will not have a high enough preemption level to join ready queue 3 and will be placed on the lowest queue at level 0 (but ahead of T1). Task T2 continues to execute. At time 8, T4 arrives but will fail the preemption rule as its deadline is not earlier than T3’s. T4 will be placed on the level 0 queue between T3 and T1.



Task T2 will continue until it completes its execution in the protected object. Its priority will then fall to 0 and T3 will preempt it. All tasks now execute in deadline order.

Final note

The above rules and descriptions are, unfortunately, not quite complete. The ready queue for Priority’First plays a central role in the model as it is, in some senses, the default queue. If a task is not entitled to be put in a higher queue, or if no protected objects are in use, then it is placed in this base queue. Indeed during the execution of a typical program most runnable tasks will be in the Priority’First ready queue most of the time. However the protocol only works if there are no protected objects with a ceiling at the Priority’First level. Such ceiling values must be prohibited, but this is not a significant constraint.

12.8.4 Example of an EDF program

Recap what the programmer needs to do to use EDF. First, preemption levels, are chosen for each task based on the temporal properties of the task. Optimally, preemption levels will be assigned in reverse (relative) deadline order. Preemption levels are assigned to tasks using the ‘priority’ attribute of each task. These will be *exactly* the same assignments that would be needed with fixed-priority assignment and the deadline-monotonic priority algorithm. Next, shared protected objects are identified and ceiling values assigned. Again this is identical in the fixed-priority and EDF schemes. Finally, the required dispatching policy is asserted.

To illustrate the minimal changes that have to be made to the application code to move from one scheduling paradigm to another consider a simple periodic task scheduled according to the standard fixed priority method. This task has a period and deadline of 10 ms.

```

task Example is
  pragma Priority(5);
end Example;

task body Example is
  Next_Release : Ada.Real_Time.Time;
  Period : Ada.Real_Time.Time_Span
    := Ada.Real_Time.Milliseconds(10);
begin
  Next_Release := Ada.Real_Time.Clock;
  loop
    -- code
    Next_Release := Next_Release + Period;
    delay until Next_Release;
  end loop;
end Example;

```

If EDF dispatching is required (perhaps to make the system schedulable) then very little change is needed. The priority level of the task remains exactly the same. The task's code must, however, now explicitly refer to deadlines (the implicit deadline was always there as it was used to derive the value 5 for the task's priority):

```

task Example is
  pragma Priority(5);
  pragma Relative_Deadline(10); -- gives an initial relative
                                -- deadline of 10 milliseconds
end Example;

task body Example is
  Next_Release: Ada.Real_Time.Time;
  Period : Ada.Real_Time.Time_Span
    := Ada.Real_Time.Milliseconds(10);
begin
  Next_Release := Ada.Real_Time.Clock;
  loop
    -- code
    Next_Release := Next_Release + Period;
    Delay_Until_and_Set_Deadline(Next_Release, Period);
  end loop;
end Example;

```

Finally the dispatching policy must be changed from

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
```

to

```
pragma Task_Dispatching_Policy(EDF_Across_Priorities);
```

No other changes are needed – any protected objects used by this or other tasks in the program retain their assigned ceilings.

12.9 Mixed scheduling

As was noted earlier, Ada allows a system with a combination of dispatching policies to be utilized. To accomplish this, the system's priority range is split into a number of distinct non-overlapping bands. In each band, a specified dispatching policy is in effect. The bands themselves are ordered by priority. So a runnable task in a high-priority band will take precedence over any other runnable task in a lower-priority band. For example, there could be a band of fixed priorities on top of a band of EDF with a single round-robin level for non-real-time tasks at the bottom. To illustrate this assume a priority range of 1..16:

```
pragma Priority_Specific_Dispatching
    (FIFO_Within_Priorities, 10, 16);

pragma Priority_Specific_Dispatching
    (EDF_Across_Priorities, 2, 9);

pragma Priority_Specific_Dispatching
    (Round_Robin_Within_Priorities, 1, 1);
```

Any task is assigned a dispatching policy by its virtue of its base priority. If a task has base priority 12 it will be dispatched according to the fixed-priority rules; if it has base priority 8 then it is under the control of the EDF rules. In a mixed system all tasks have a preemption level (it is just its base priority) and all tasks have a deadline (it will be `Default_Deadline` if none is assigned in the program). However, this deadline will have no effect if the task is not in an EDF band. With this example a runnable task with priority 14 will always execute before an 'EDF task' even if the 'EDF task' has an earlier deadline.

To achieve any mixture including one or more EDF bands, the properties assigned in the definition of EDF dispatching to level `Priority_First` need to be redefined to use the minimum value of whatever range is used for the EDF tasks. Also note that two adjacent EDF bands are not equivalent to one complete band. Runnable tasks in the upper bands will always take precedence over runnable tasks in the lower bands.

For completeness any priority value not included in the pragma is assumed to be `FIFO_Within_Priorities`. Any of the predefined policies can be mixed apart from the non-preemptive one. It is deemed incompatible to mix non-preemption with any other scheme as non-preemption is a system-wide property; a low-priority preemptive task would impact on a higher-priority task in another band.

Tasks within different bands can communicate using protected objects (and rendezvous if required). The use of `Ceiling_Locking` ensures that protected objects behave as required. For example an 'EDF task' (priority level 7) could share data with a 'round-robin' task (priority level 1) and a 'fixed-priority' task (priority 12). The protected object would have a ceiling value of (at least) 12. When the EDF task accesses the object its active priority will rise from 7 to 12, and while executing this protected action it will prevent any other task executing from within the EDF band. The 'round-robin' task will similarly execute with priority 12 – if its quantum is exhausted inside the object it will continue to execute until it has completed the protected action.

If a task changes its base priority at run-time (using the `Set_Priority` routine) then it may also change its dispatching group. For example, a task in the above illustration with base priority 7 that is changed to have priority 12 will move from EDF to fixed-priority scheduling.

Perhaps one minor weakness of Ada's support for mixed dispatching rules is that a task cannot directly ask under what policy it is being scheduled. However, a task can always find out its own priority (using `Get_Priority`) and from that use program constants to ascertain under which policy it is executing.

This ability to mix dispatching policies is unique to Ada. Experience will show whether this level of support for real-time programs proves to be useful, and if implementations are able to deliver this flexibility in an efficient manner.

Summary

For scheduling theory to be employed in real applications it must be accessible from the programming languages used to implement the applications. Either the language must directly support the scheduling schemes or it must provide a means by which the application can access these schemes as they are supported by the underlying operating system.

The most mature scheduling theory is based upon preemptive fixed-priority dispatching. In this chapter the means by which Ada, C/Real-Time POSIX and Real-Time Java support this dispatching scheme have been described. A common set of primitives is evident in all three languages. This bears witness to the maturing of the theory.

However, fixed-priority dispatching is not the only scheduling scheme available. Earliest Deadline First (EDF) is one of a number of alternative schemes that are, theoretically, at least as important. Unfortunately EDF is not as easily available to application programmers. Unless a bespoke EDF-oriented operating system is constructed, the only way to make use of EDF is via the provisions now available in Ada. This chapter has therefore also described Ada's support for EDF scheduling. This has included discussions of its use of a variety of Baker's algorithms for controlling access to protected objects. Finally in this chapter the means by which mixed dispatching schemes (e.g. fixed-priority, EDF and round-robin) can be programmed in Ada have been included.

Further reading

- Burns, A. and Wellings, A.J. (2007) *Concurrent and Real-Time Programming in Ada 2005*. Cambridge: Cambridge University Press.
- Butenhof, D. R. (1997) *Programming With Posix Threads*. Reading, MA: Addison-Wesley.
- Hyde, P. (1999) *Java Thread Programming*. Indianapolis, IN: Sams Publishing.
- Lea, D. (1999) *Concurrent Programming in Java: Design Principles and Patterns*. Reading, MA: Addison-Wesley.

- Oaks, A. and Wong, H. (1997) *Java Threads*. Sebastopol, CA: O'Reilly.
- Nichols, B., Buttlar, D. and Farrell, J. (1996) *POSIX Threads Programming*. Sebastopol, CA: O'Reilly.
- Wellings, A.J. (2004) *Concurrent and Real-Time Programming in Java*. Chichester: Wiley.

Exercises

- 12.1** A real-time systems designer wishes to run a mixture of safety-critical, mission-critical and non-critical periodic and sporadic Ada tasks on the same processor. He or she is using preemptive priority-based scheduling and has used the response time analysis equation to predict that all tasks meet their deadlines. Give reasons why the system might nevertheless fail to meet its deadlines at run-time. What enhancements could be provided to the Ada run-time support systems to help eliminate the problems?
- 12.2** Ada allows the base priority of a task to be set dynamically. Using the `Ada.Dynamic_Priorities` package, show how to implement a mode change protocol where a group of tasks must have their priorities changed as a single atomic operation.
- 12.3** Explain the pros and cons of supporting dynamic ceiling priorities.
- 12.4** Show how earliest deadline first scheduling can be implemented in Real-Time Java with a feasibility test of total process utilization less than 100%.

Chapter 13

Tolerating timing faults

13.1	Dynamic redundancy and timing faults	13.5	Overrun of resource usage
13.2	Deadline miss detection	13.6	Damage confinement
13.3	Overrun of worst-case execution time	13.7	Error recovery
13.4	Overrun of sporadic events		Summary
			Further reading
			Exercises

Throughout this book it has been assumed that real-time systems have high reliability requirements. One method of achieving this reliability is to incorporate fault tolerance into the software. The inclusion of timing constraints introduces the possibility of these constraints being broken at run-time and failures occurring in the time domain. With soft systems, a task may need to know if a timing constraint has been missed, even though it can accommodate this under normal execution. ^{More importantly, in a hard system (or soft system), where deadlines are critical, a missed deadline needs to trigger some error recovery routine.}

If the system has been shown to be schedulable under worst-case execution times then it is arguable that deadlines cannot be missed. However, the discussions of reliability in Chapter 2 indicated strongly the need for a multifaceted approach to reliability; that is, prove that nothing can go wrong and include routines for adequately dealing with the problems that arise when they do.

This chapter considers the causes of timing faults and how they can be tolerated within the context of the dynamic redundancy approach to fault tolerance introduced in Chapter 2.

13.1 Dynamic redundancy and timing faults

In Chapter 2, the four phases of dynamic software fault tolerance were introduced. These are now reviewed in the context of timing faults.

- (1) **Error detection** – Most timing faults will eventually manifest themselves in the form of missed deadlines.
- (2) **Damage confinement and assessment** – When deadlines have been missed, it must be decided which tasks in the system are at fault. Ideally confinement techniques should ensure that only faulty tasks miss their deadlines.

- (3) **Error recovery** – The response to deadline misses requires that the application undertakes some recovery, perhaps providing a degraded service.
- (4) **Fault treatment and continued service** – Timing errors often result from transient overloads. Hence they can often be ignored. However, persistent deadline misses may indicate more serious problems and require some form of maintenance to be undertaken.

In a system that has been ‘proved’ correct in the timing domain via the schedulability analysis techniques presented in Chapter 11, deadlines could still be missed if:

- worst-case execution time (WCET) calculations were inaccurate (optimistic rather than pessimistic);
- blocking times were underestimated;
- assumptions made in the schedulability checker were not valid;
- the schedulability checker itself had an error;
- the scheduling algorithm could not cope with a load even though it is theoretically schedulable;
- the system is working outside its design parameters, for example sporadic events occurring more frequently than was assumed in the schedulability analysis.

In this latter case (for instance, an information overflow manifesting itself as an unacceptable rate of interrupts), the system designers may still wish for fail-soft or fail-safe behaviour.

Chapter 2 introduced the *fault* \rightarrow *error* \rightarrow *failure* chain. Assuming the schedulability analysis is correct, the following chains are possible in the context of priority-based systems (dos Santos and Wellings, 2008).

- (1) *Fault* (in task τ_i ’s WCET calculation or assumptions) \rightarrow *error* (overrun of τ_i ’s WCET) \rightarrow *error propagation* (deadline miss of τ_i) \rightarrow *failure* (to deliver service in a timely manner).
- (2) *Fault* (in task τ_i ’s WCET calculation or assumptions) \rightarrow *error* (overrun of τ_i ’s WCET) \rightarrow *error propagation* (greater interference on lower-priority tasks) \rightarrow *error propagation* (deadline miss of lower priority tasks) \rightarrow *failure* (to deliver service in a timely manner).
- (3) *Fault* (in task τ_i ’s minimum inter-arrival time assumptions) \rightarrow *error* (greater computation requirement for τ_i) \rightarrow *error propagation* (deadline miss of τ_i) \rightarrow *failure* (to deliver service in a timely manner).
- (4) *Fault* (in task τ_i ’s minimum inter-arrival time assumptions) \rightarrow *error* (greater interference on lower priority tasks) \rightarrow *error propagation* (deadline miss of lower-priority tasks) \rightarrow *failure* (to deliver service in a timely manner).
- (5) *Fault* (in task τ_i ’s WCET calculation or assumptions when using a shared resource) \rightarrow *error* (overrun of τ_i ’s resource usage) \rightarrow *error propagation* (greater blocking time of higher-priority tasks sharing the resource) \rightarrow *error propagation* (deadline miss of higher-priority tasks) \rightarrow *failure* (to deliver service in a timely manner).

Similar chains will exist for other scheduling approaches and faults, where instead of the term ‘lower/higher priority’ the corresponding eligibility criterion can be substituted (e.g. ‘later/earlier absolute deadline’).

To be tolerant of timing faults, it is necessary to be able to detect:

- miss of a deadline – the final error in all the above error propagation chains;
- overrun of a worst-case execution time – potentially causing the task and/or lower eligibility tasks to miss their deadlines (error chains 1 and 2);
- a sporadic event occurring more often than predicted – potentially causing the task and/or lower eligibility tasks to miss their deadlines (error chains 3 and 4);
- overrun in the usage of a resource – potentially causing higher eligibility tasks to miss their deadlines (error chain 5).

Of course the last three error conditions do not necessarily indicate that deadlines will be missed; for example, an overrun of WCET in one task might be compensated by a sporadic event occurring less often than the maximum allowed. Hence, the damage confinement and assessment phase of providing fault tolerance (introduced in Section 2.5) must determine what actions to take. Both forward and backward recovery are then possible.

If timing faults are to be handled, their associated error conditions have to be detected first. If the run-time environment or operating system is aware of the salient characteristics of a task (as it is in the Real-Time Java approach, for example), it will be able to detect problems and bring them to the attention of the application. Alternatively, it is necessary to provide primitive facilities that will allow the application to detect its own timing errors. With all error detection mechanisms, the earlier the problem can be detected the more chance there is to pinpoint the problem and the more time there is to recover.

The following sections will discuss error detection mechanisms for the above timing faults, consider possible error confinement approaches, and identify strategies for recovery. Fault treatment typically involves maintenance, which is a topic outside the scope of this book. However, it is noted that for non-stop real-time applications some form of dynamic change management or mode change is required. The impact of this can have serious real-time implications.

13.2 Deadline miss detection

Deadline miss detection is the minimum that is required if a real-time system is to tolerate timing failures. It is a ‘catch all’ mechanism that will detect even problems outside the failure hypothesis. For example, it will detect problems resulting from errors in the schedulability analysis. Of course, with all ‘catch all’ mechanisms, it may be difficult to pinpoint the cause of the problem and it leaves little time for recovery.

This section discusses how deadline miss detection is facilitated in Ada, Real-Time Java and C/Real-Time POSIX. Strategies for dealing with deadline misses are considered in Section 13.7.

13.2.1 Ada

Ada 2005 allows the deadline of a task to be specified and this parameter can be used to influence scheduling (see Section 12.8). However, the Ada run-time support system

does not use this information to detect deadline misses. For this, the language provides primitive mechanisms that the programmer has to use to detect the missed deadline itself. One way of achieving this is to use the asynchronous transfer of control facility discussed in Section 7.6.1. Hence to detect a deadline overrun of the periodic task given in Section 10.2 requires the main functionality of the task to be embedded in a ‘select then abort’ statement.

```

task body Periodic_T is
  Next_Release : Time;
  Next_Deadline : Time;
  Release_Interval : constant Time_Span := Milliseconds(...);
  Deadline : constant Time_Span := Milliseconds(...);
begin
  -- read clock and calculate the next
  -- release time (Next_Release) and
  -- next deadline (Next_Deadline)
  loop
    select
      delay until Next_Deadline;
      -- deadline miss detected here
      -- perform recovery
    then abort
      -- sample data (for example) or
      -- calculate and send a control signal
    end select;
    delay until Next_Release;
    Next_Release := Next_Release + Release_Interval;
    Next_Deadline := Next_Release + Deadline;
  end loop;
end Periodic_T;

```

A similar approach can be used to detect a deadline miss in a sporadic task.

One of the problems with this approach is that it combines detection with a particular recovery strategy, that of stopping the task from what it is doing. This is, clearly, one option. Another is to use a different task to handle the deadline miss. The possible recovery strategies could include extending the deadline, lowering the errant task’s priority, or some other action short of terminating it (see Section 13.7).

To simply detect a deadline miss, the Ada timing event facility that was discussed in Section 10.4 can be used (see Program 10.6). In Section 2.5.1, the watchdog timer approach to fault detection was presented. This can be easily programmed using Ada’s timing events. The essential idea is that on initializing the watchdog with a first deadline and a period, the watchdog sets up a timing event for the first deadline. The task must now call the watchdog to reset the timing event before its deadline has expired. If it does so, the watchdog sets the event to expire at the next deadline, and so on. If the task does not call to reset the event, the event will be triggered.

The specification of the watchdog is given first:

```

protected type Watchdog(Event : access Timing_Event) is
  procedure Initialize(First_Deadline : Time;
                     Required_Period : Time_Span);
  entry Alarm_Control(T: out Task_Identity);
  -- Called by alarm handling task

```



```

procedure Call_In;
    -- Called by application code when it completes.

    pragma Interrupt_Priority (Interrupt_Priority'Last);
private
    procedure Timer(Event : in out Timing_Event);
        -- Timer event code, ie the handler.
    Alarm : Boolean := False;
    Tid : Task_Identifier;
    Next_Deadline : Time;
    Period : Time_Span;
end Watchdog;

```

This watchdog object has a common structure. An entry with an initially closed barrier holds back a monitoring task that will be released by the handler if the handler executes. In this example the handler is actually never executed unless there is a missed deadline. Each time the monitored task calls `Call_In`, the timing event is reset to a point in the future. Only if another call does not occur before its next deadline will the handler be executed and the barrier opened releasing the monitoring task.

```

protected body Watchdog is
    procedure Initialize(First_Deadline : Time;
                        Required_Period : Time_Span) is
    begin
        Next_Deadline := First_Deadline;
        Period := Required_Period;
        Set_Handler(Event.all, Next_Deadline, Timer'Access)
        Tid = Current_Task;
    end Initialize;

    entry Alarm_Control(T: out Task_Identity) when Alarm is
    begin
        T := Tid;
        Alarm := False;
    end Alarm_Control;

    procedure Timer(Event : in out Timing_Event) is
    begin
        Alarm := True;
        -- Note no use is made of the parameter in this example
    end Timer;

    procedure Call_in is
    begin
        Next_Deadline := Next_Deadline + Period;
        Set_Handler(Event.all, Next_Deadline, Timer'Access);
        -- Note this call to Set_Handler cancels the previous call
    end Call_in;
end Watchdog;

```

The revised structure of the periodic task is shown below:

```

with Watchdogs; use Watchdogs;
...
Watch : Watchdog;

```

```

Deadline_Miss_Event : aliased Timing_Event;
Set_Handler(Deadline_Miss_Event, Next_Deadline, Timer'Access);

task body Periodic_T is
  Next_Release : Time;
  Release_Interval : constant Duration := ...; -- or
  Release_Interval : constant Time_Span := Milliseconds(...);
begin
  -- read clock and calculate the next
  -- release time (Next_Release)
  -- and first deadline (First_Deadline)
  Watch.Initialize(First_Deadline, Release_Interval);
  loop
    -- sample data (for example) or
    -- calculate and send a control signal
    Watch.Call_In;
    delay until Next_Release;
    Next_Release := Next_Release + Release_Interval;
  end loop;
end Periodic_T;

```

13.2.2 Real-Time Java

Unlike the Ada run-time system, the Real-Time Java virtual machine does monitor the deadlines of real-time threads and will release asynchronous event handlers when periodic or sporadic tasks are still executing when their deadlines have passed, the handlers (`missHandler`) being identified with the release parameters associated with the real-time threads (see Program 10.3).

The full semantics of Real-Time Java's deadline miss detection are somewhat complex. Program 13.1 shows the associated methods. Recall the structure of a period real-time thread from Chapter 10:

```

public class Periodic extends RealtimeThread {
  public Periodic(PeriodicParameters P)
  { ... };

  public void run() {
    boolean deadlineMet = true;
    while(deadlineMet) {
      // code to be run each period
      ...
      deadlineMet = waitForNextPeriod();
    }
  }
}

```

An aperiodic or sporadic thread has a similar structure only with a call to `waitForNextRelease` instead of `waitForNextPeriod`.

The full semantics can be summarized by the following points.

- If the real-time thread misses its deadline, and it has an associated deadline miss handler, this is released at the point the deadline expires. The real-time thread is

Program 13.1 An extract of the `RealtimeThread` class showing methods used for deadline miss detection.

```
package javax.realtime;
public class RealtimeThread extends Thread
    implements Schedulable {
    ...

    // the following methods are used with periodic execution
    public boolean waitForNextPeriod();
    public void deschedulePeriodic();
        // deschedules the periodic thread at the end
        // of its current release
    public void schedulePeriodic();
        // reschedules the periodic thread at its next release event

    // used for aperiodic and sporadic execution
    public boolean waitForNextRelease();
    public void deschedule();
        // deschedules the aperiodic thread at the end
        // of its current release
    public void schedule();
        // reschedules the aperiodic thread at its next release event
    ...
}
```

automatically de-scheduled – this means that at the end of its current release (when it calls `waitForNextPeriod/waitForNextRelease`), the scheduler will no longer consider the thread for possible execution until it has been explicitly rescheduled by the application via a call to `schedulePeriodic/schedule`. At this point the thread becomes eligible for execution at its next release event.

- If there is no associated handler, when the deadline miss occurs a count (called `deadlineMiss`) of the number of missed deadlines is incremented.
- The `waitForNextPeriod` method (wFNP) has the following semantics (`waitForNextRelease` has similar semantics):
 - If no deadlines have been missed, wFNP returns true when its next release event occurs.
 - When the `deadlineMiss` count is greater than zero and the previous call to wFNP returned true, wFNP decrements the `deadlineMiss` count and returns false immediately. This situation indicates that the current release has missed its deadline. At this point, the current release is still active.
 - When the `deadlineMiss` count is greater than zero and the previous call to wFNP returned false, wFNP decrements the `deadlineMiss` count and returns false immediately. This situation indicates that the next release time has already passed and the next deadline has already been missed. At this point, the current release has completed and the next release is active.

- When a deadline miss handler has been released and the `deadlineMiss` count equals zero and no call to the `schedulePeriodic` method has occurred since the deadline miss handler was released, `wfNP` de-schedules the real-time thread until an explicit call to the `schedulePeriodic` method occurs (probably by the released handler); `wfNP` then returns true at the point of the next release after the call to `schedulePeriodic`. At this point, the next release is active.

An example of using the Real-Time Java facilities will be given in Section 13.7 where strategies for recovery are considered.

13.2.3 C/Real-Time POSIX

C/Real-Time POSIX allows timers to be created and set which will generate user-defined signals (`SIGALRM` by default) when they expire. Hence this will allow the process to decide what is the correct course of action to pursue. Program 10.7 shows a typical C interface.

The watchdog timer approach to fault detection can be easily programmed using POSIX signals. For example, consider the case where one thread (`monitor`) creates another thread (`server`) and wishes to monitor its progress to see if it meets a deadline. The deadline of the `server` is given by `struct timespec deadline`. The `monitor` thread creates a per process timer indicating a signal handler to be executed if the timer expires. It then creates the `server` thread and passes a pointer to the timer. The `server` thread performs its action and then deletes the timer. If the alarm goes off, the `server` is late.

```
#include <signal.h>
#include <timer.h>
#include <pthread.h>

timer_t timer; /* timer shared between monitor and server */

struct timespec deadline = ...;
struct timespec zero = ...;

struct itimerspec alarm_time, old_alarm;

struct sigevent s;

void server(timer_t *watchdog) {
    /* perform required service */
    TIMER_DELETE(*watchdog);
}

void watchdog_handler(int signum, siginfo_t *data,
                      void *extra)
{
    /* SIGALRM handler */

    /* server is late: undertake recovery */
}
```



```

void monitor() {
    pthread_attr_t attributes;
    pthread_t serve;

    sigset_t mask, omask;
    struct sigaction sa, osa;
    int local_mode;

    SIGEMPTYSET(&mask);
    SIGADDSET(&mask, SIGALRM);

    sa.sa_flags = SA_SIGINFO;
    sa.sa_mask = mask;
    sa.sa_sigaction = &watchdog_handler;

    SIGACTION(SIGALRM, &sa, &osa); /* assign handler */

    alarm_time.it_value = deadline;
    alarm_time.it_interval = zero; /* one shot timer */

    s.sigev_notify = SIGEV_SIGNAL;
    s.sigev_signo = SIGALRM;

    TIMER_CREATE(CLOCK_REALTIME, &s, &timer);

    TIMER_SETTIME(timer, TIMER_ABSTIME, &alarm_time, &old_alarm);

    PTHREAD_ATTR_INIT(&attributes);
    PTHREAD_CREATE(&serve, &attributes, (void *)server, &timer);
}

```

However, as noted in Section 7.5.1, if a process is multithreaded, the signal is sent to the whole process, not an individual thread. In general, therefore, to generalize the above approach it is necessary to pass a value with the signal being generated to indicate the associated thread (via the `sigev_value` component of the `sigevent` structure).

13.2.4 Timing error detection at the block level

Task-level deadlines are the most common deadlines. However, as noted in Section 9.6, deadlines can occur at a finer granularity, for example at the block level. The watchdog approach given in the previous sections allows this block-level detection to be achieved. However, neither Ada, Real-Time Java or C/Real-Time POSIX provides direct support for block-level deadlines. The research language DPS considered in Section 10.6.3 illustrates the type of support that could be given.

In DPS, timing errors are associated with exceptions:

```

start <timing constraints> do
    -- statements
exception

```

```
-- handlers
end
```

In addition to the necessary computations required for damage limitation, error recovery and so on, the handler may wish to extend the deadline period and continue execution of the original block. Thus a *resumption* rather than *termination* model may be more appropriate (see Chapter 3).

In a time-dependent system, it may also be necessary to give the deadline constraints of the handlers. Usually the execution time for the handler is taken from the temporal scope itself; for example, in the following, the statement sequence will be prematurely terminated after 19 time units:

```
start elapse 22 do
  -- statements
exception
  when elapse_error within 3 do
    -- handler
end
```

As with all exception models, if the handler itself gives rise to an exception this can only be dealt with at a higher level within the program hierarchy. If a timing error occurs within a handler at the task level then the task must be terminated (or at least the current iteration of the task). There might then be some system-level handlers to deal with failed tasks or it may be left to the application software to recognize and cope with such events.

If exception handlers are added to the coffee-making example given with DPS in Section 10.6.3, the code would have the following form (exceptions for logic errors such as ‘no cups available’ are not included). It is assumed that only `boil_water` and `drink_coffee` have any significant temporal properties; timing errors are, therefore, due to overrun on these activities.

```
from 9:00 to 16:15 every 45 do
  start elapse 11 do
    get_cup
    boil_water
    put_coffee_in_cup
    put_water_in_cup
  exception
    when elapse_error within 1 do
      turn_off_kettle -- for safety
      report_fault
      get_new_cup
      put_orange_in_cup
      put_water_in_cup
    end
  end
end

start after 3 elapse 26 do
  drink
exception
  when elapse_error within 1 do
```



```

        empty_cup
    end
end
replace_cup
exception
    when any_exception do
        null    -- go on to next iteration
    end
end
end

```

13.3 Overrun of worst-case execution time

Good fault tolerance practices attempt to confine the consequences of an error to a well-defined region of the program. Facilities such as modules, packages and atomic actions help with this goal. However, if a task consumes more of the CPU resource than has been anticipated, then it may not be that task that misses its deadline. For example, in the case of a high-priority task with a fair amount of slack time, the tasks that will miss their deadlines may be lower-priority tasks with less slack available. Ideally, it should be possible to catch the timing error in the task that caused it. This implies that it is necessary to be able to detect when a task overruns the worst-case execution time that the implementer has allowed for it. Of course, if a task is non-preemptively scheduled (and does not block waiting for resources), its CPU execution time is equal to its elapse time and the same mechanisms that were used to detect deadline overrun can be used. However, tasks are usually preemptively scheduled, and this makes measuring CPU time usage more difficult. It usually has to be supported explicitly in the host operating system.

13.3.1 Execution-time clocks in Real-Time POSIX

POSIX supports execution-time monitoring using its clock and timer facilities. Two clocks are defined: `CLOCK_PROCESS_CPUTIME_ID` and `CLOCK_THREAD_CPUTIME_ID`. These can be used in the same way as `CLOCK_REALTIME`. Each process/thread has an associated execution-time clock; calls to:

```

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &some_timespec_value);

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &some_timespec_value);

clock_getres(CLOCK_PROCESS_CPUTIME_ID, &some_timespec_value);

```

will set/get the execution-time or get the resolution of the execution-time clock associated with the calling process (similarly for threads).

Two functions allow a process/thread to obtain and then access the clock of another process/thread.

```

int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);

int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id);

```

The timers defined in Program 10.7 can be used to create timers which, in turn, can be used to generate process-signals when the execution time set has expired. It is implementation-defined what happens if a `timer_create` is used with a `clock_id` different from that of the calling process/thread. As the signal generated by the expiry of the timer is directed at the process, it is application-dependent which thread will get the signal if a thread's execution-time timer expires. An application can disallow the use of the timer for a thread (because of the overhead in supporting the facility).

As with all execution-time monitoring, it is difficult to guarantee the accuracy of the execution-time clock in the presence of context switches and interrupts.

13.3.2 Execution-time clocks in Ada

Ada 2005 directly supports execution-time clocks for tasks, and supports timers that can be fired when tasks have used a defined amount of execution time. Indeed it has added a clock per task that measures the task's execution time. A package, `Ada.Execution_Time` – see Program 13.2, is defined that is similar in structure to `Ada.Calendar` and `Ada.Real_Time`.

Program 13.2 An abridged version of the `Ada.Execution_Time` package.

```
with Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;

package Ada.Execution_Time is
  type CPU_Time is private;
  CPU_Time_First : constant CPU_Time;
  CPU_Time_Last  : constant CPU_Time;
  CPU_Time_Unit  : constant :=
    <implementation-defined-real-number>;
  CPU_Tick : constant Time_Span;

  function Clock
    (T : Ada.Task_Identification.Task_ID ...
     := Ada.Task_Identification.Current_Task)
    return CPU_Time;

  function "+" (Left : CPU_Time; Right : Time_Span) return CPU_Time;
  function "+" (Left : Time_Span; Right : CPU_Time) return CPU_Time;
  -- similarly for "-", "<", "<=", ">" and ">="

  procedure Split
    (T : CPU_Time; SC : out Seconds_Count; TS : out Time_Span);

  function Time_Of (SC : Seconds_Count; TS : Time_Span)
    return CPU_Time;

  ...
private
  -- Not specified by the language.
end Ada.Execution_Time;
```

When a task is created so is an execution-time clock. This clock registers zero at creation and starts recording the task execution time from the point at which the task starts its *activation* (see Section 4.4). To read the value of any task's clock a `Clock` function is defined. So in the following a loop is allocated 7 ms of execution time before exiting at the end of its current iteration:

```
Start : CPU_Time;
Interval : Time_Span := Milliseconds(7);
...

Start := Ada.Execution_Time.Clock;
while Ada.Execution_Time.Clock - Start < Interval loop
  --code
end loop;
```

Note the '-' operator returns a value of type `Time_Span` which can then be compared with `Interval`.

To monitor the execution time of each invocation of a periodic task, for example, is simple:

```
Last : CPU_Time;
Exe_Time : Time_Span;
Last := Execution_Time.Clock;
loop
  -- code of task
  Exe_Time := Ada.Execution_Time.Clock - Last;
  Last := Ada.Execution_Time.Clock;
  -- print out or store Exe_Time
  delay until ...
end loop;
...
```

As well as monitoring a task's execution time profile, it is also possible to trigger an event if its execution-time clock gets to some specified value. A child package of `Ada.Execution_Time` provides support for this type of event – see Program 13.3.

Each `Timer` event is strongly linked to the task that will trigger it. This static linkage is ensured by the access discriminant for the type that is required to be **constant** and **not null**.

The handler type is standard, but there is now a need to specify the minimum ceiling priority the associated protected object must have if ceiling violation is to be avoided. This priority will be set by the supporting implementation.

The `Set_Handler` procedures and the other routines all have the same properties as those defined with timing events. However, in recognition that an implementation may have a limited capacity for timers, or that only one timer per task is possible, the exception `Timer_Resource_Error` may be raised when a `Timer` is defined or when a `Set_Handler` procedure is called for the first time with a new `Timer` parameter.

An example of using the Ada facilities will be given in Section 13.7 where strategies for recovery are considered.

Program 13.3 The Ada.Execution.Time.Timers package.

```

package Ada.Execution_Time.Timers is

    type Timer(T : not null access constant
               Ada.Task_Identification.Task_ID) is tagged
               limited private;

    type Timer_Handler is access protected
        procedure(TM : in out Timer);

    Min_Handler_Ceiling : constant System.Any_Priority :=
        <Implementation Defined>;

    procedure Set_Handler(TM: in out Timer;
                          In_Time : Time_Span; Handler : Timer_Handler);
    procedure Set_Handler(TM: in out Timer;
                          At_Time : CPU_Time; Handler : Timer_Handler);

    procedure Cancel_Handler(TM : in out Timer;
                             Cancelled : in out Boolean);

    function Current_Handler(TM : Timer) return Timer_Handler;

    function Time_Remaining(TM : Timer) return Time_Span;

    Timer_Resource_Error : exception;

private
    -- Not specified by the language.
end Ada.Execution_Time.Timers;

```

13.3.3 Execution-time monitoring in Real-Time Java

Unlike, Ada and Real-Time POSIX, Real-Time Java does not support explicitly CPU-time clocks. Instead it allows a ‘cost’ value to be associated with the execution of a schedulable object as one of the attributes in the ReleaseParameters class (see Program 10.3). This is some measure of how much of the processor’s time is required to execute the computation associated with the real-time thread’s release (that is, after it has been released and until it has completed). An optional facility can be supported by the real-time virtual machine that keeps track of the CPU time consumed by the thread on each release. An asynchronous event can be fired if the real-time thread consumes more than this CPU time.

Real-Time Java (as of RTSJ Version 1.1) also allows information on the amount of execution time consumed by a real-time thread during its releases to be obtained via the following methods in the RealtimeThread class:

```

RelativeTime getMaxConsumption()
RelativeTime getMaxConsumption(RelativeTime dest);
// Returns the maximum amount of CPU time this

```



```
// schedulable object has used in a single release
// in a newly-allocated RelativeTime object or
// in an instance supplied by the caller.
```

```
RelativeTime getMinConsumption()
RelativeTime getMinConsumption(RelativeTime dest)
// Returns the minimum amount of CPU time this
// schedulable object has used in a single release
// in a newly-allocated RelativeTime object or
// in an instance supplied by the caller.
```

Real-Time Java does not require that an implementation monitor the processing time consumed by schedulable objects, as this requires support from the underlying operating system. Many operating systems do not currently provide this support.

If cost monitoring is supported, then Version 1.02 of Real-Time Java requires that the priority scheduler gives a real-time thread a CPU budget of no more than its cost value on each release. Hence, if a real-time thread overruns its cost budget, it is automatically de-scheduled (made not eligible for execution) immediately. It will not be rescheduled until either its next release occurs (in which case its budget is replenished) or its associated cost value is increased. In Version 1.1 of Real-Time Java, this will be the default policy. However, there will also be a mechanism that allows just notification rather than enforcement.

13.4 Overrun of sporadic events

A sporadic event firing more frequently than anticipated can have an enormous impact on a system attempting to meet hard deadlines. Here the term **sporadic overrun** is used to denote this fault. Where the event is the result of an interrupt, the consequences can be potentially devastating. For example, during the first landing on the moon, the guidance computer was reset after a CPU on the Lunar Landing Module was flooded with radar data interrupts. The landing was nearly aborted as a result (Regehr, 2007).

There are a variety of techniques that have been developed over the years to deal with this situation. The two basic approaches are either to stop the early firing from occurring, or to bound the total amount of the CPU time allocated to all events from the same source. The latter use ‘server’ technology which will be discussed in detail in Section 13.6.1 in the context of damage confinement. The features provided by C/Real-Time POSIX will also be considered there.

Here the focus is on what support can be provided to prohibit the firings or to detect them when they occur and take some corrective action. Two types of events are considered: those resulting from hardware interrupts and those resulting from the software firing of an event.

13.4.1 Handling sporadic event overruns in Ada

In keeping with the overall Ada philosophy, low-level mechanisms can be used to handle event overruns.

Where the event is triggered by a hardware interrupt, on most occasions the interrupt can be inhibited from occurring by manipulating the associated device control registers (see Chapter 14). A simple approach is illustrated below. Here, the assumption is that there is a required minimum inter-arrival time (MIT) between interrupt occurrences.

```
protected Sporadic_Interrupt_Controller is
  procedure Interrupt; -- mapped onto interrupt
  entry Wait_For_Next_Interrupt;
private
  procedure Timer(Event : in out Timing_Event);
  Call_Outstanding : Boolean := False;
  MIT : Time_Span := Milliseconds(...);
end Sporadic_Interrupt_Controller;

Event: Timing_Event;

protected body Sporadic_Interrupt_Controller is
  procedure Interrupt is
  begin
    -- turn off interrupts
    Set_Handler(Event, MIT, Timer'Access);
    Call_Outstanding := True;
  end Interrupt;

  entry Wait_For_Next_Interrupt when Call_Outstanding is
  begin
    Call_Outstanding := False;
  end Wait_For_Next_Interrupt;

  procedure Timer(Event : in out Timing_Event) is
  begin
    -- Turn interrupts back on
  end Timer;
end Sporadic_Interrupt_Controller;
```

Once an interrupt from the device occurs, interrupts are disabled and a timing event is set to expire at the required minimum inter-arrival time. Once this occurs, the device's interrupts are enabled.

The sporadic task has a familiar structure:

```
task body Sporadic_T is
begin
  loop
    Sporadic_Interrupt_Controller.Wait_For_Next_Interrupt;
    -- action
  end loop;
end Sporadic_T;
```

Of course, it is device dependent what happens if the device wants to interrupt and is unable to. Usually, the device overruns and any data is lost.

If the event is fired from a software task, then the above approach can be modified so that (for example) an exception is raised.


```

protected Sporadic_Interrupt_Controller is
  procedure Release; -- mapped onto interrupt
  entry Wait_For_Next_Release;
private
  Call_Outstanding : Boolean := False;
  MIT : Time_Span := Milliseconds(...); -- Minimum inter-arrive time
  Last_Release : Time;
end Sporadic_Interrupt_Controller;

MIT_VIOLATION : exception;

protected body Sporadic_Interrupt_Controller is
  procedure Release is
    Now : Time := Clock;
  begin
    if Now - Last_Release < MIT then
      raise MIT_VIOLATION;
    else
      Last_Release := Now;
    end if;
    Call_Outstanding := True;
  end Interrupt;

  entry Wait_For_Next_Release when Call_Outstanding is
  begin
    Call_Outstanding := False;
  end Wait_For_Next_Interrupt;
end Sporadic_Interrupt_Controller;

```

Raising an exception is only one possible approach.

The usual constraint on a sporadic task is that there is a minimum separation between any two releases. A generalization of this constraint, which allows for bursts of release events, is to set a limit of M release in any length of time L . Although these constraints are a little more complicated to program, the above approaches can be extended to this M in L case.

13.4.2 Real-Time Java and minimum inter-arrival time violations

The previous subsection illustrated how to handle various MIT violation conditions using Ada. In keeping with Ada's philosophy, it is up to the program to detect these conditions and act accordingly.

The Real-Time Java philosophy is the opposite. It provides the mechanisms that allow the real-time JVM to detect MIT violation. The `SporadicParameters` class defines a subclass of release parameters that allow the programmer to specify that a real-time thread is a sporadic thread. The class is shown in Program 13.4.

The following policies are available.

- `mitViolationExcept` – an exception is thrown in the releasing real-time thread. If the violation is on an asynchronous event handler released by the firing of an interrupt, then the policy defaults to the `mitViolationIgnore` policy.

Program 13.4 An abridged version of the SporadicParameters class.

```

package javax.realtime;
public class SporadicParameters
    extends AperiodicParameters {
    // fields
    public static final String mitViolationExcept;
    public static final String mitViolationIgnore;
    public static final String mitViolationReplace;
    public static final String mitViolationSave;

    public SporadicParameters(RelativeTime minInterarrival);
    public SporadicParameters(RelativeTime minInterarrival,
        RelativeTime cost, RelativeTime deadline,
        AsyncEventHandler overrunHandler,
        AsyncEventHandler missHandler);

    // methods
    public String getMitViolationBehavior();
    public void setMitViolationBehavior(String behavior);
    public RelativeTime getMinimumInterarrival();
    public void setMinimumInterarrival(
        RelativeTime interarrival);

    ...
}

```

- mitViolationIgnore – the release event is ignored.
- mitViolationReplace – the last release event is overwritten with the current one.
- mitViolationSave – the release event is delayed until the MIT has passed.

It perhaps should be noted that in Real-Time Java interrupt handlers are second level interrupt handlers. The program cannot rely on the above approach to prohibit the interrupt from occurring. To do this, it must adopt the approach illustrated in Ada, and turn off interrupts. It is also not possible for the programmer to specify other more general constraints, such as *M in L* discussed above.

13.5 Overrun of resource usage

Problems caused by errors in accessing resources are notoriously difficult to handle. At a functional level, they can corrupt shared state and potentially lead to deadlock. From a timing perspective, the whole *raison d'être* for priority (or more generally, eligibility) inheritance protocols such as those discussed in Section 11.8 is to avoid timing problems. However, even approaches involving inheritance and ceiling protocols can cause problems if the blocking time assumed by the schedulability analysis is incorrect. There are two main potential causes for this:

- a task may overrun its allotted access time for the resource; or
- unanticipated resource contention that has not been taken into account in the blocking-time analysis.

The latter is possible in large systems with the use of prewritten library code. In Section 9.4 timeouts were introduced as a mechanism for detecting the absence of some expected communication or event. There are several reasons why these are inadequate in the context of blocking time.

- (1) With inheritance protocols, blocking is cumulative. Timeout could be used on entry to critical sections; however, the programmer would have to keep a running track of the total blocking time in the current release.
- (2) With immediate ceiling protocols, the blocking occurs before execution of the task. Hence a timeout has no use, as there is no contention when accessing the critical section.
- (3) Support for critical sections does not always have a timeout mechanism. For example, both Java's synchronized methods and Ada protected types have no associated timeout mechanisms. Although C/Real-Time POSIX provides a timed version of `mutex_lock`, no indication of how long the calling task was blocked for is returned.

Hence, whilst timeouts have a role to play, their use is limited in this context.

For finer control, it is possible to use detection of WCET overruns at the block level. Hence all resource accesses would have to be policed to ensure that the calling task did not overrun its allotted usage. Of course, as already pointed out in Section 13.1, an overrun in one resource may be compensated by underuse of another. Furthermore, detecting overruns on every resource access may be prohibitively expensive.

As a last resort, overruns in blocking times will, if significant, cause tasks to miss their deadlines, which will be detected. Good programming practice dictates that synchronized code is short and of a simple form. Errors are therefore far less likely.

13.6 Damage confinement

The role of damage confinement of time-related faults is to prevent propagation of the resulting errors to other components in the system. There are two aspects of this that can be identified:

- protecting the system from the impact of sporadic task overruns and unbounded aperiodic activities;
- supporting composability and temporal isolations.

The problem of overruns in sporadic objects has already been mentioned in Section 13.4. Aperiodic events also present a similar problem. As they have no well-defined release characteristics, they can impose an unbounded demand on the processor's time. If not handled properly, they can result in periodic or sporadic tasks missing their deadlines,

even though those tasks have been ‘guaranteed’. In Section 11.6.2, **aperiodic servers** were introduced. Aperiodic servers protect the processing resources needed by periodic tasks but otherwise allow aperiodic and sporadic tasks to run as soon as possible. Several types of servers were discussed including the Sporadic Server and Deferrable Server.

When composing systems of multiple applications, whether dynamically or statically, it is often required that each application be isolated from one another. Memory management hardware has provided that isolation in the spatial domain for many years. However, the facility to support temporal isolation, where the applications share the same computing resource, has only recently become available. This has been brought about by **hierarchy schedulers** and **reservation-based systems**. Usually, two levels of scheduling are used; a global (top-level) scheduler and multiple application-level (second-level) schedulers. Typically the application-level scheduler is also called a **server** or **execution-time server** or **group server**. The latter term will be used in this book.

Although the above confinement techniques are similar, they have slightly different emphases. For temporal isolation, the key requirement is that the group server be *guaranteed its budget each period*; that is, it must be possible for the tasks contained within the group to consume all the group’s budget on each release (and not be allowed to consume any more). To support aperiodic execution, it is sufficient that the aperiodic server *consumes no more than its budget each period*. Hence schedulability analysis can be undertaken on tasks scheduled within a group. Whereas, the analysis of the impact of an aperiodic server can be bounded, typically no analysis of the tasks contained within the server need be done. If a group contains only sporadic tasks, the budget must be guaranteed. The goal here is to ensure that the sporadic task does not violate the CPU time that has been allocated to it, for example by being released more often than its minimum inter-arrival time and consuming more than its maximum budget on each release.

With group servers, the schedulability analysis is simpler if the associated tasks are ‘bound’. The term ‘bound’ in this context refers to the relationship between the tasks’ periods and the period of the group server. A bound relationship is where the periods of the tasks are exact multiples of the period of the group and have arrival times that coincide with its replenishment.

13.6.1 Programming servers with C/Real-Time POSIX

C/Real-Time POSIX supports the Sporadic Server approach to damage confinement as one of the scheduling policies (see Program 12.2). The policy can be applied at both the thread and process levels. As discussed in Section 11.6.2, a Sporadic Server assigns a limited amount of CPU capacity to handle events, and has a replenishment period, a budget and two priorities. The server runs at a high priority when it has some budget left and a low one when its budget is exhausted. When a server runs at the high priority, the amount of execution time it consumes is subtracted from its budget. The amount of budget consumed is replenished at the time the server was activated plus the replenishment period. When its budget reaches zero, the server’s priority is set to the low value.

At the thread level, the Sporadic Server provides confinement for sporadic and aperiodic activities. Each sporadic thread is assigned a Sporadic Server scheduling policy, and is given appropriate parameters. Note that these say nothing about the minimum

Program 13.5 A typical abridged C interface to support the POSIX sporadic server facilities.

```
#define SCHED_SPORADIC ... /* sporadic server */
#define PTHREAD_SCOPE_SYSTEM ... /* system-wide contention */
#define PTHREAD_SCOPE_PROCESS ... /* local contention */

typedef ... pid_t;
struct sched_param {
    ...
    timespec sched_ss_repl_period
    timespec sched_ss_init_budget
    int sched_ss_max_repl
    ...
};

int sched_setparam(pid_t pid, const struct sched_param *param);
/* set the scheduling parameters of process pid */

int sched_get_priority_max(int policy);
/* returns the maximum priority for the policy specified */

int sched_get_priority_min(int policy);
/* returns the minimum priority for the policy specified */

int pthread_attr_setscope(pthread_attr_t *attr,
                          int contentionscope);
/* set the contention scope for a thread attribute object */

int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *param);
/* set the scheduling policy for a thread attribute object */

/* All the above integer functions return 0 if successful */
/* There are similar setter methods */
```

inter-arrival time; instead they bound the impact that the thread can have to be equivalent to a periodic activity whose characteristics are that of the server's parameters (shown in Program 13.5). Handling aperiodic activities with this approach is more problematic as it not possible to assign the budget to a group of threads. For them it is necessary to use a Sporadic Server *process*.

A Sporadic Server process is a process that is scheduled according to the sporadic server policy. Hence *all* the threads contained within it share the allocated budget. Hence, timing errors are confined to those threads. C/Real-Time POSIX also supports shared memory objects between processes; hence it is possible to partition a single application between processes and still communicate via shared memory (there is an attribute to POSIX mutexes that caters for this option). If the scheduling has system-wide contention, then the threads will compete at their own priority. If the scheduling has local contention, then the process's priority will be the dominating factor.

13.6.2 Programming servers with Real-Time Java

Real-Time Java provides support for temporal isolation via:

- an optional cost monitoring and enforcement model – see Section 13.3.3;
- sporadic release parameters – see Section 13.4.2;
- processing group parameters – considered in this subsection.

Real-Time Java provides support for group servers via **processing groups**. A processing group is implicitly created when an instance of the `ProcessingGroupParameter` class is created. When processing group parameters are assigned to one or more aperiodic schedulable objects, a server is effectively created. The server's start time, cost (capacity) and period are defined by the particular instance of the parameters. These collectively define the points in time when the server's capacity is replenished. Any aperiodic schedulable object that belongs to a processing group is executed at its own defined priority. However, it only executes if the server still has capacity. As it executes, each unit of CPU time consumed is subtracted from the server's capacity. When the capacity is exhausted, the aperiodic schedulable objects are not allowed to execute until the start of the next replenishment period. If the application only assigns aperiodic schedulable objects of the same priority level to a single `ProcessingGroupParameters` object, then the functionality of a Deferrable Server can be obtained.

Real-Time Java is, however, a little more general. It allows schedulable objects of different priorities to be assigned to the same group, the inclusion of sporadic and periodic schedulable objects, the 'servers' to be given a deadline, and cost overrun and deadline miss handlers to be set. This represents an extensive set of facilities.

If used within the context of an aperiodic server, a cost overrun potentially indicates a transient overload where the aperiodic load cannot be handled effectively. Of course, the tasks will be executed across one or more of the following aperiodic server's periods, but this will impact on their response times. For a Deferrable Server, the deadline would equal the period and deadline misses are not relevant at the server level.

In the context of a group server (execution-time server), a cost overrun is potentially more severe. It indicates that the workload assigned to the group is an underestimate, and consequently some deadlines of the threads may be missed.¹ If the group itself has missed its deadline then the guaranteed capacity given to the server has been violated. The `ProcessingGroupParameters` class is given below in Program 13.6.

Although Real-Time Java does not define any feasibility analysis, the `setIfFeasible` method allows the Processing Group to be guaranteed.

13.6.3 Programming servers in Ada

Although, Ada does not directly support servers, it does provide the primitives from which servers can be programmed.

¹Real-Time Java supports deadline overrun detection on an individual real-time thread/asynchronous event handler basis.

Program 13.6 The ProcessingGroupParameter class.

```

package javax.realtime;
public class ProcessingGroupParameters
    implements Cloneable {

    public ProcessingGroupParameters(
        HighResolutionTime start, RelativeTime period,
        RelativeTime cost, RelativeTime deadline,
        AsyncEventHandler overrunHandler,
        AsyncEventHandler missHandler);

    // methods
    public Object clone();
    public RelativeTime getCost();
    public AsyncEventHandler getCostOverrunHandler();
    public RelativeTime getDeadline();
    public AsyncEventHandler getDeadlineMissHandler();
    public RelativeTime getPeriod();
    public HighResolutionTime getStart();

    public void setCost(RelativeTime cost);
    public void setCostOverrunHandler(
        AsyncEventHandler handler);
    public void setDeadline(RelativeTime deadline);
    public void setDeadlineMissHandler(
        AsyncEventHandler handler);
    public void setPeriod(RelativeTime period);
    public void setStart(HighResolutionTime start);

    public boolean setIfFeasible(RelativeTime period,
        RelativeTime cost, RelativeTime deadline);
}

```

Group budgets allow different group servers to be implemented; consequently the language itself does not have to provide a small number of predefined server types. Note, servers can be used with fixed-priority or EDF scheduling.

A typical server has a budget and a replenishment period. At the start of each period, the available budget is restored to its maximum amount. Unused budget at this time is discarded. To program a server requires timing events to trigger replenishment and a means of grouping tasks together and allocating them an amount of CPU resource. A standard package (a child of `Ada.Execution_Time` – see Program 13.7) is defined to accomplish this. The type `Group_Budget` represents a CPU budget to be used by a group of tasks.

There are a number of routines defined in this package. Consider first those concerned with the grouping of tasks. Each `Group_Budget` has a set of tasks associated with it. Tasks are added to the set by calls of `Add_Task`, and removed using `Remove_Task`. Functions are defined to test if a task is a member of any group budget,

Program 13.7 The Ada.Execution_Time.Group_Budgets package.

```

package Ada.Execution_Time.Group_Budgets is
  type Group_Budget is tagged limited private;

  type Group_Budget_Handler is access
    protected procedure(GB : in out Group_Budget);

  type Task_Array is array(Positive range <>) of
    Ada.Task_Identification.Task_ID;

  Min_Handler_Ceiling : constant System.Any_Priority :=
    <Implementation Defined>;

  procedure Add_Task(GB: in out Group_Budget;
    T : Ada.Task_Identification.Task_ID);
  procedure Remove_Task(GB: in out Group_Budget;
    T : Ada.Task_Identification.Task_ID);
  function Is_Member(GB: Group_Budget;
    T : Ada.Task_Identification.Task_ID) return Boolean;
  function Is_A_Group_Member(
    T : Ada.Task_Identification.Task_ID) return Boolean;
  function Members(GB: Group_Budget) return Task_Array;

  procedure Replenish(GB: in out Group_Budget; To : Time_Span);
  procedure Add(GB: in out Group_Budget; Interval : Time_Span);
  function Budget_Has_Expired(GB: Group_Budget) return Boolean;
  function Budget_Remaining(GB: Group_Budget) return Time_Span;

  procedure Set_Handler(GB: in out Group_Budget;
    Handler : Group_Budget_Handler);
  function Current_Handler(GB: Group_Budget)
    return Group_Budget_Handler;
  procedure Cancel_Handler(GB: in out Group_Budget;
    Cancelled : out Boolean);

  Group_Budget_Error : exception;
private
  -- not specified by the language
end Ada.Execution_Time.Group_Budgets;

```

or one specific group budget. A further function returns the collection of tasks associated with a group budget by returning an unconstrained array type of task IDs.

An important property of these facilities is that a task can be a member of at most one group budget. Attempting to add it to a second group will cause Group_Budget_Error to be raised. When a task terminates, if it is still a member of a group budget, it is automatically removed.

The budget decreases whenever a task from the associated set executes. The accuracy of this accounting is again implementation defined. To increase the amount of budget available, two routines are provided. The Replenish procedure sets the budget

to the amount of 'real-time' given in the `To` parameter. It replaces the current value of the budget. By comparison, the `Add` procedure increases the budget by the `Interval` amount but, as this parameter can be negative it can also be used to, in effect, reduce the budget.

To inquire about the state of the budget, two functions are provided. Note that when `Budget_Has_Expired` returns `True` then `Budget_Remaining` will return `Time_Span_Zero`.

A handler is associated with a group budget by use of the `Set_Handler` procedure. There is an implicit event associated with a `Group_Budget` that occurs whenever the budget goes to zero. If at that time there is a non-null handler set for the budget, the handler will be executed.

As with timers, an implementation must define the minimum ceiling priority level for the protected object linked to any group budget handler. Also note there are `Current_Handler` and `Cancel_Handler` subprograms defined.

By comparison with timers and timing events, which are triggered when a certain clock value is reached (but will then never be reached again for monotonic clocks), the group budget event can occur many times – whenever the budget goes to zero. So the handler is permanently associated with the group budget, and it is executed every time the budget is exhausted (obviously following replenishment and further usage). The handler can be changed by a further call to `Set_Handler` or removed by using a null parameter to this routine (or by calling `Cancel_Handler`), but for normal execution the same handler is called each time. The better analogy for a group budget event is an interrupt; its handler is called each time the interrupt occurs.

When the budget is zero, the associated tasks *continue* to execute. If action should be taken when there is no budget, this has to be programmed (it must be instigated by the handler). So group budgets are not in themselves a server abstraction – but they allow these abstractions to be constructed.

To give a simple example, consider four aperiodic tasks that should share a budget of 2 ms that is replenished every 10 ms. The tasks first register with a `Controller1` protected object that will manage the budget. They then loop around waiting for the next invocation event. In all of the examples in this section, fixed-priority scheduling on a single processor is assumed.

```
task Aperiodic_Task is
  pragma Priority(Some_Value);
end Aperiodic_Task;

task body Aperiodic_Task is
  ...
begin
  Controller1.Register;
  loop
    -- wait for next invocation
    -- undertake the work of the task
  end loop;
end Aperiodic_Task;
```

The `Controller1` protected object will use a timer event and a group budget, and hence defines handlers for both.

```

protected Controller1 is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  entry Register;
  procedure Timer_Handler(E : in out Timing_Event);
  procedure Group_Handler(G : in out Group_Budget);
private
  T_Event : Timing_Event;
  G_Budget : Group_Budget;
  For_All : Boolean := False;
end Controller1;

protected body Controller1 is
  entry Register when Register'Count = 4 or For_All is
  begin
    if not For_All then
      For_All := True;
      G_Budget.Add(Milliseconds(2));
      G_Budget.Add_Task(Register'Caller);
      T_Event.Set_Handler(Milliseconds(10), Timer_Handler'Access);
      G_Budget.Set_Handler(Group_Handler'access);
    else
      G_Budget.Add_Task(Register'Caller);
    end if;
  end Register;

  procedure Timer_Handler(E : in out Timing_Event) is
    T_Array : Task_Array := G_Budget.Members;
  begin
    G_Budget.Replenish(Milliseconds(2));
    for ID in T_Array'Range loop
      Asynchronous_Task_Control.Continue(T_Array(ID));
    end loop;
    E.Set_Handler(Milliseconds(10), Timer_Handler'Access);
  end Timer_Handler;

  procedure Group_Handler(G : in out Group_Budget) is
    T_Array : Task_Array := G.Members;
  begin
    for ID in T_Array'Range loop
      Asynchronous_Task_Control.Hold(T_Array(ID));
    end loop;
  end Group_Handler;
end Controller1;

```

The Register entry blocks all calls until each of the four 'clients' has called in. The final task to register (which becomes the first task to enter) sets up the group budget and the timing event, adds itself to the group and alters the boolean flag so that the other three tasks will also complete their registration. For these tasks it is straightforward to add themselves to the group budget. Note the tasks in this example may have different priorities.

The two handlers work together to control the tasks. Whenever the group budget handler executes, it stops the tasks from executing by using the Hold routine. It always gets a new list of members in case any have terminated. The Timer_Handler releases

all the tasks using `Continue`, it replenishes the budget and then sets up another timing event for the next period (10 ms).

In a less stringent application it may be sufficient to just prevent new invocations of each task if the budget is exhausted. The current execution is allowed to complete and hence tasks are not suspended. The following example implements this simpler scheme, and additionally allows tasks to register dynamically (rather than all together at the beginning). The protected object is made more general-purpose by representing it as a type with discriminants for its main parameters (replenishment in terms of milliseconds and budget measured in microseconds):

```
protected type Controller2(Period, Bud : Positive) is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  procedure Register;
  entry Proceed;
  procedure Timer_Handler(E : in out Timing_Event);
  procedure Group_Handler(G : in out Group_Budget);
private
  T_Event : Timing_Event;
  G_Budget : Group_Budget;
  First : Boolean := True;
  Allowed : Boolean := False;
  Req_Budget : Time_Span := Microseconds(Bud);
  Req_Period : Time_Span := Milliseconds(Period);
end Controller2;

Con : Controller2(10, 2000);
```

The client task would now have the following structure:

```
task body Aperiodic_Task is
  ...
begin
  Con.Register;
  loop
    -- wait for next invocation
    -- undertake the work of the task
  end loop;
end Aperiodic_Task;
```

The body of the controller is as follows:

```
protected body Controller2 is
  entry Proceed when Allowed is
  begin
    null;
  end Proceed;

  procedure Register is
  begin
    if First then
      First := False;
      Add(G_Budget, Req_Budget);
      T_Event.Set_Handler(Req_Period, Timer_Handler'Access);
```

```

    G_Budget.Set_Handler(Group_Handler'Access);
    Allowed := True;
end if;
G_Budget.Add_Task(Current_Task);
end Register;

procedure Timer_Handler(E : in out Timing_Event) is
begin
    Allowed := True;
    G_Budget.Replenish(Req_Budget);
    E.Set_Handler(Req_Period, Timer_Handler'Access);
end Timer_Handler;

procedure Group_Handler(G : in out Group_Budget) is
begin
    Allowed := False;
end Group_Handler;
end Controller2;

```

The next example illustrates a Deferable Server. Here, the server has a fixed priority, and when the budget is exhausted, the tasks are moved to a background priority *Priority'First*. This is closer to the first example, but retains some of the properties of the second approach:

```

protected type Controller3(Period, Bud : Positive;
                           Pri : Priority) is
    pragma Interrupt_Priority (Interrupt_Priority'Last);
    procedure Register;
    procedure Timer_Handler(E : in out Timing_Event);
    procedure Group_Handler(G : in out Group_Budget);
private
    T_Event : Timing_Event;
    G_Budget : Group_Budget;
    First : Boolean := True;
    Req_Budget : Time_Span := Microseconds(Bud);
    Req_Period : Time_Span := Milliseconds(Period);
end Controller3;

Con : Controller3(10, 2000, 12);
-- assume this server has priority 12

protected body Controller3 is
    procedure Register is
    begin
        if First then
            First := False;
            G_Budget.Add(Req_Budget);
            T_Event.Set_Handler(Req_Period, Timer_Handler'Access);
            G_Budget.Set_Handler(Group_Handler'Access);
        end if;
        Add_Task(G_Budget, Current_Task);
        if G_Budget.Budget_Has_Expired then
            Set_Priority(Priority'First);
            -- sets client task to background priority

```



```

    else
        Set_Priority(Pri);
        -- sets client task to servers 'priority'
    end if;
end Register;

procedure Timer_Handler(E : in out Timing_Event) is
    T_Array : Task_Array := G_Budget.Members;
begin
    G_Budget.Replenish(Req_Budget);
    for ID in T_Array'Range loop
        Set_Priority(Pri, T_Array(ID));
    end loop;
    E.Set_Handler(Req_Period, Timer_Handler'Access);
end Timer_Handler;

procedure Group_Handler(G : in out Group_Budget) is
    T_Array : Task_Array := G_Budget.Members;
begin
    for ID in T_Array'Range loop
        Set_Priority(Priority'First, T_Array(ID));
    end loop;
end Group_Handler;
end Controller3;

```

When a task registers, it is running outside the budget so it is necessary to check if the budget is actually exhausted during registration. If it is then the priority of the task must be set to the low value. Other properties of this algorithm should be clear to the reader from the previous discussions.

13.7 Error recovery

Once timing errors have been detected, strategies for recovery must be developed. Inevitably this is application-dependent; however, there are several techniques that can be utilized. This section first considers recovery at the individual thread/task level, and then considers more system-wide responses.

13.7.1 Task-level recovery

The goal of the damage-confinement techniques outlined in the previous section has been to attempt to isolate timing errors to individual tasks or groups of tasks.

Strategies for handling WCET overrun

Monitoring WCET overrun has been suggested as a mechanism for detecting a common fault before the error propagates outside the errant task. Once detected, the task response will depend on whether it is a hard, soft or firm.

- **WCET overrun in hard real-time tasks** – although the error detection techniques introduced in Chapter 2 have detected functional failures that might cause overruns (such as non-terminating loops), WCET overrun can still occur due to inaccuracies in calculating the WCET values. One possibility is that the WCET values used in the schedulability analysis consist of the addition of two components. The first is the time allocated for the primary algorithm and the second is the time for recovery (assuming a fault hypothesis of a single failure per task per release). The first time is the time that is used by the system when monitoring. When this time passes, forward or backward error recovery occurs and the alternative algorithm is executed. This can either be within the same task and the budget increased (for example, changing the cost in a Real-Time Java thread's release parameters), or by releasing a dedicated recovery task. Typically, these alternative algorithms try to provide a degraded service. Another possibility is simply to do nothing. This assumes that there is enough slack in the system for the task (and other lower-priority tasks) to still meet their deadlines.
- **WCET overruns in soft/firm real-time tasks** – typically overruns in soft and firm real-time tasks can be ignored if the isolation techniques guarantee the capacity needed for the hard real-time tasks. Alternatively, the tasks' priorities can be lowered, or the current releases can be terminated and the tasks re-released when their next release event occurs.

As an example of the latter, consider the use of Ada's execution-time timers and timer events to lower the priority of a task if it overruns its worst-case execution time. In this example, the task has a worst-case execution time of 1.25 ms per invocation. If it executes for more than this value its priority should be lowered from its correct value of 14 to a minimum value of 2. If it is still executing after a further 0.25 ms then that invocation of the task must be terminated; this implies the use of an ATC construct. First, the overrun handler protected type is defined:

```
protected Overrun is
  pragma Priority(Min_Handler_Ceiling);
  entry Stop_Task;
  procedure Handler(TM : in out Timer);
  procedure Reset(C1 : CPU_Time);
private
  Abandon : Boolean := False;
  First_Occurrence : Boolean := True;
  WCET_Overrun : CPU_Time;
end Overrun;

protected body Overrun is
  entry Stop_Task when Abandon is
  begin
    null;
  end Stop_Task;

  procedure Reset(C1 : CPU_Time) is
  begin
    Abandon := False;
    First_Occurrence := True;
```



```

    WCET_Overrun := C1;
end Reset;

procedure Handler(TM : in out Timer) is
begin
    if First_Occurrence then
        Set_Handler(TM,WCET_Overrun,Handler'Access);
        Set_Priority(2, TM.T.all);
        First_Occurrence := False;
    else
        Abandon := True;
    end if;
end Handler;
end Overrun;

```

It may not be immediately clear why a Reset routine is required, but without it a race condition may lead to incorrect execution. Consider the code of the task:

```

task Hard_Example;
task body Hard_Example is
    ID : aliased Task_ID := Current_Task;
    WCET_Error : Timer(ID'access);
    WCET : CPU_Time := Ada_Execution_Time.Time_Of(0,
                                                Microseconds(1250));
    WCET_Overrun : CPU_Time := Time_Of(0,Microseconds(250));
    Bool : Boolean := False;
    ...
begin
    -- initialization
    loop
        Overrun.Reset(WCET_Overrun);
        Set_Handler(WCET_Error,WCET,Overrun.Handler'Access);
        select
            Overrun.Stop_Task;
            -- handler the error if possible at priority level 2
        then abort
            -- code of the application
        end select;
        Cancel_Handler(WCET_Error, Bool);
        Set_Priority(14);
        delay until ...
    end loop;
    ...
end Hard_Example;

```

It is possible for the timer to trigger (or *expire*) after completion of the select statement but before it can be cancelled. This would leave the state of the boolean variable, Abandon, with the incorrect value of True. Similarly, it is necessary to cancel the timer before changing the priority back to 14 – otherwise the event could trigger just before executing the delay statement and the task would be stuck with the wrong low priority for its next invocation.

Strategies for handling sporadic event overruns

There are several responses to the violation of minimum inter-arrival time of a sporadic task. The mechanism provided by Real-Time Java covers most of them: the release event can be ignored, an exception can be thrown, the last event can be overwritten (if it has not already been acted upon) or the actual release of the thread can be delayed until the MIT has passed. Of course, the thread could ignore the violation and be executed anyway.

Strategies for handling deadline misses

Although the early identification of potential timing problems facilitates damage assessment, many real-time systems just focus on the recovery from missed deadlines. Again, several strategies are possible.

- **Deadline miss of hard real-time tasks** – it is possible to set two deadlines for each task. An early deadline is one whose miss will cause the invocation of forward or backward error recovery. A later deadline is the deadline used by the schedulability test. In both cases, the recovery should again aim to produce a degraded service for the task.
- **Deadline miss of soft real-time task** – typically this can be ignored and treated as a transient overload situation. A count of missed deadlines can be maintained, and when it passes a certain threshold a health monitoring system can be informed (see below).
- **Deadline miss of a firm real-time task** – as a firm task produces no value once it has passed its deadline, its current release can be terminated.

As an example of handling a deadline miss, consider a soft real-time Real-Time Java system where applications will want to monitor any deadline misses, but take no action unless a certain threshold is reached. When it is reached, the tardy thread is de-scheduled.

Here, a health monitor object is assumed with the following interface:

```
import javax.realtime.*;
public class HealthMonitor {
    public void persistentDeadlineMiss(Schedulable s);
}
```

Now consider the following event handler for catching a missed deadline of a periodic real-time thread:

```
import javax.realtime.*;
class DeadlineMissHandler extends AsyncEventHandler {
    public DeadlineMissHandler(HealthMonitor mon,
                               int threshold) {
        super(new PriorityParameters(
            PriorityScheduler.MAX_PRIORITY),
            null, null, null, null, null);
        myHealthMonitor = mon;
    }
}
```



```

    myThreshold = threshold;
}

public void setThread(RealtimeThread rt) {
    myrt = rt;
}

public void handleAsyncEvent() {
    if(++missDeadlineCount < myThreshold)
        myrt.schedulePeriodic();
    else
        myHealthMonitor.persistentDeadlineMiss(myrt);
}
private RealtimeThread myrt;
private int missDeadlineCount = 0;
private HealthMonitor myHealthMonitor;
private final int myThreshold;
}

```

When the handler is executed, it increments the miss count and reschedules the thread. When the count reaches the threshold, it informs the health monitor and does not reschedule the thread. The following code sets up the real-time periodic thread.

```

{
    PriorityScheduler ps = (PriorityScheduler)Scheduler.
        getDefaultScheduler();
    HealthMonitor healthMonitor = new HealthMonitor();
    DeadlineMissHandler missHandler = new
        DeadlineMissHandler(healthMonitor, 5);
    PriorityParameters ppl = new
        PriorityParameters(ps.getMinPriority());
    PeriodicParameters release1 = new PeriodicParameters(
        new RelativeTime( 0,0),    // start,
                                new RelativeTime(1000,0), // period
                                new RelativeTime( 100,0), // cost
                                new RelativeTime(500, 0), // deadline
                                null,                      // no overrun handler
                                missHandler);              // miss handler

    RealtimeThread rtt1 = new RealtimeThread(ppl,release1) {
        public void run() {
            // Code for thread.
        }
    }
    missHandler.setThread(rtt1);
    rtt1.start();
}

```

Of course, the deadline miss detection mechanism can be combined with the Real-Time Java ATC mechanism to stop the task if necessary.

13.7.2 Mode changes and event-based reconfiguration

In the above discussions, it has generally been assumed that a missed deadline and other timing errors can be dealt with by the task that is actually responsible for the problem. This is not always the case. Often the consequences of a timing error are as follows.

- Other tasks must alter their deadlines or even terminate what they are doing.
- New tasks may need to be started.
- Critically important computation may require more processor time than is currently available; to obtain the extra time, other less significant tasks may need to be 'suspended'.
- Tasks may need to be 'interrupted' in order to undertake one of the following (typically):
 - immediately return their best results they have obtained so far;
 - change to quicker (but presumably less accurate) algorithms;
 - forget what they are presently doing and become ready to take new instructions: 'restart without reload'.

These actions are sometimes known as **event-based reconfiguration**.

Some systems may additionally enter anticipated situations in which deadlines are liable to be missed. A good illustration of this is found in systems that experience **mode changes**. This is where some event in the environment occurs which results in certain computations that have already started, no longer being required. If the system were to complete these computations then other deadlines would be missed; it is thus necessary to terminate prematurely the tasks or temporal scopes that contain the computations.

To perform event-based reconfiguration and mode changes requires communication between the tasks concerned. Due to the asynchronous nature of this communication, it is necessary to use the asynchronous notification mechanisms found in languages like Ada, Real-Time Java and C/Real-Time POSIX (see Section 7.4). These mechanisms are low level; arguably what is really required is to be able to tell the scheduler to stop invoking certain threads that are now not required and to begin to invoke other tasks. Real-Time Java goes some way towards this by having methods associated with real-time threads that inform the scheduler that the real-time thread is currently not required. The methods are shown in Program 13.1. De-scheduling and rescheduling real-time threads in Real-Time Java does not alter the phasing of the thread. It simply ignores any release event for the thread. Also note, the real-time thread is allowed to complete its current release.

The research language Real-Time Euclid adopts a slight different approach from Real-Time Java because it ties its asynchronous event-handling mechanism to its real-time release mechanisms. In Real-Time Euclid time constraints are associated with processes (a task is called a process in Real-Time Euclid) and numbered exceptions can be defined. Handlers must be provided in each process. For example, consider the following temperature controller process which defines three exceptions.


```

process TempController : periodic frame 60 first activation
                        atTime 600 or atEvent startMonitoring
% import list
handler (except_num)
  exceptions (200,201,304) % for example
  imports (var consul, ...)
  var message : string(80), ...
  case except_num of
    label 200: % very low temperature
      message := "reactor is shut down"
      consul := message
    label 201: % very high temperature
      message := "meltdown has begun - evacuate"
      consul := message
      alarm := true % activate alarm device
    label 304: % timeout on sensor
              % reboot sensor device

  end case
end handler
%
% execution part
%
end TempController

```

Real-Time Euclid allows a process to raise an exception in another process. Three different kinds of raise statement are supported: *except*, *deactivate* and *kill*; as their names imply they have increasing severity.

The *except* statement is essentially the same as the Ada and Java raise/throw statements, the difference being that once the handler has been executed, control is returned to where it left off (that is, the resumption model). By comparison, the *deactivate* statement causes that iteration of the (periodic) process to be terminated. The victim process still executes the exception handler, but will then only become reactivated when its next period is due. Hence Real-time Euclid allows the current release to be abandoned.

To terminate a process, the *kill* statement is available; this explicitly removes a process (possibly itself) from the set of active processes. It differs from an unconditional abort in that the exception handler is executed before termination. This has the advantage that a process may perform some important 'last rites'. It has the disadvantage that an error in the handler could still cause the process to malfunction.

To illustrate the use of these exceptions, the temperature control process given in Section 10.6.1 will have some detail added to its execution part. Note that, in this example, exceptions are raised and handled synchronously within the same process, and asynchronously in another process. First, the process waits on a condition variable; a timeout is specified and an exception number is given. (If this timeout occurs, the numbered exception is raised using *except*.) A temperature is then read and logged. Tests on the temperature value could lead to other exceptions being raised. A low value will result in an appropriate message and deactivation until the next period; a high value will result in an even more appropriate, if somewhat futile, message, an exception being raised in an alarm process, and the temperature controller terminating. All available processor time can now be dedicated to the alarm process.

```

process TempController : periodic frame 60 first activation
                        atTime 600 or atEvent startMonitoring

% import list
handler (except_num)
  exceptions (200,201,304) % for example
  imports (var consul, ...)
  var message : string(80), ...
  case except_num of
    ... % as before
  end case
end handler

wait(temperature_available) noLongerThan 10 : 304
currentTemperature := ... % low-level i/o
log := currentTemperature
if currentTemperature < 100 then
  deactivate TempController : 200
elseif currentTemperature > 10000 then
  kill TempController : 201
end if
% other computations
end TempController

```

To perform this form of reconfiguration in Ada, two mechanisms are available:

- abort – similar to *kill*
- ATC – similar to *deactivate*.

Ada allows ‘last rites’ to be programmed using a *controlled* variable. As indicated in Section 7.6.1, the ATC feature is a general one, and hence it can deal with *deactivate* and most other forms of event-based reconfiguration.

Summary

This chapter has discussed the toleration of timing faults from within the framework of dynamic software fault tolerance. Timing faults manifest themselves in the following conditions:

- overrun of deadline;
- overrun of worst-case execution time;
- sporadic events occurring more often than predicted;
- timeouts on communications.

Ada and Real-Time POSIX provide low-level mechanisms that the programmer can use to detect these conditions, whereas Real-Time Java provides support in the context of real-time threads and release parameters.

Execution time and aperiodic servers provide the main mechanisms in support of damage confinement. Ada and Real-Time Java provide this via the

notion of group budgets. Ada, in particular, has a flexible set of mechanisms that allow various server approaches to be implemented. Real-Time POSIX opts for the support of a single policy, that of a sporadic server.

Error recovery strategies depend on an application's context. Many timing errors can be considered transient and can be ignored. Others require a task to stop what it is doing and undertake an alternative action. On occasions, a task in isolation can not deal with the problem, and reconfiguration and mode changes may need to be performed.

In some high-integrity applications, there is little scope for error handling within the program itself. A deadline miss will lead to an attempt to recover at the system level. This could involve a switch to another version of the software running on another processor, or the cold restarting of the current program.

Further reading

Burns, A. and Wellings, A. J. (2007) *Concurrent and Real-Time Programming in Ada*. Cambridge: Cambridge University Press.

Koptez, H. (1997) *Real-time Systems*. New York: Kluwer Academic.

Wellings, A. J. (2004) *Concurrent and Real-Time Programming in Java*. Chichester: Wiley.

Exercises

- 13.1 Show how Ada can implement block-level deadline violations. How can the equivalent be done in Real-Time Java or Real-Time POSIX?
- 13.2 Can the deadline overrun of a periodic thread in Real-Time Java be detected?
- 13.3 What is the role of deadline in the specification of a Real-Time Java event handler?
- 13.4 To what extent can the violation of maximum arrival frequency of sporadic events be detected in Ada, Real-Time POSIX and Real-Time Java?
- 13.5 To what extent can event-based reconfiguration be performed in Real-Time Java?
- 13.6 Outline (with code) how Ada supports sporadic tasks. How can the task protect itself from executing more often than its minimum inter-arrival time?
- 13.7 To what extent can sporadic servers be implemented in Ada?
- 13.8 To what extent can `ProcessingGroupParameters` be used by a Real-Time Java scheduler to support sporadic servers?

Chapter 14

Low-level programming

14.1	Hardware input/output mechanisms	14.5	C and older real-time languages
14.2	Language requirements	14.6	Scheduling device drivers
14.3	Ada	14.7	Memory management
14.4	Real-Time Java		Summary
			Further reading
			Exercises

One of the main characteristics of an embedded system is the requirement that it interacts with special-purpose input and output (I/O) devices. Unfortunately, there are many different types of device interfaces and control mechanisms. This is mainly because: different computers provide different methods of interfacing with devices; different devices have separate requirements for their control; and similar devices from different manufacturers have different interfaces and control requirements.

To provide a rationale for the high-level language facilities needed to program low-level devices, one must understand the basic hardware I/O functions. Therefore this chapter considers these mechanisms first, then deals with language features in general, and finally gives details of particular languages.

Embedded systems often have limited memory capacity. Consequently, the programmer must ensure that the compiler allocates only the memory needed for the job in hand. Furthermore, dynamic memory allocation must be carefully controlled to ensure that time-critical code is not preempted by the inopportune execution of system functions such as a garbage collector.

14.1 Hardware input/output mechanisms

As far as input and output devices are concerned, there are two general classes of computer architecture: one with a logically separate bus for memory and I/O and the other with memory and I/O devices on the same logical bus. These are represented diagrammatically in Figures 14.1 and 14.2.

The interface to a device is normally through a set of **device registers**. With logically separate buses for memory and I/O devices, the computer must have two sets of

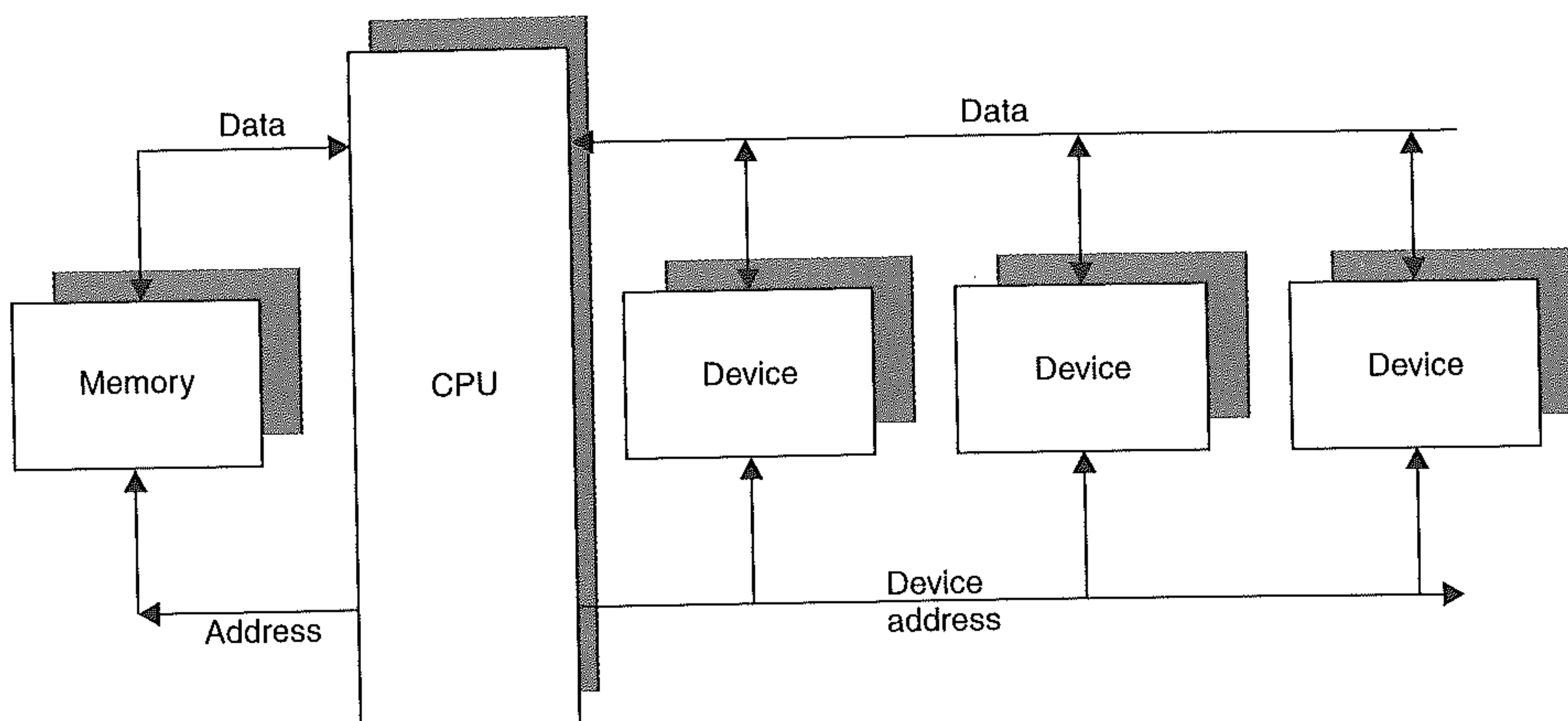


Figure 14.1 Architecture with separate buses.

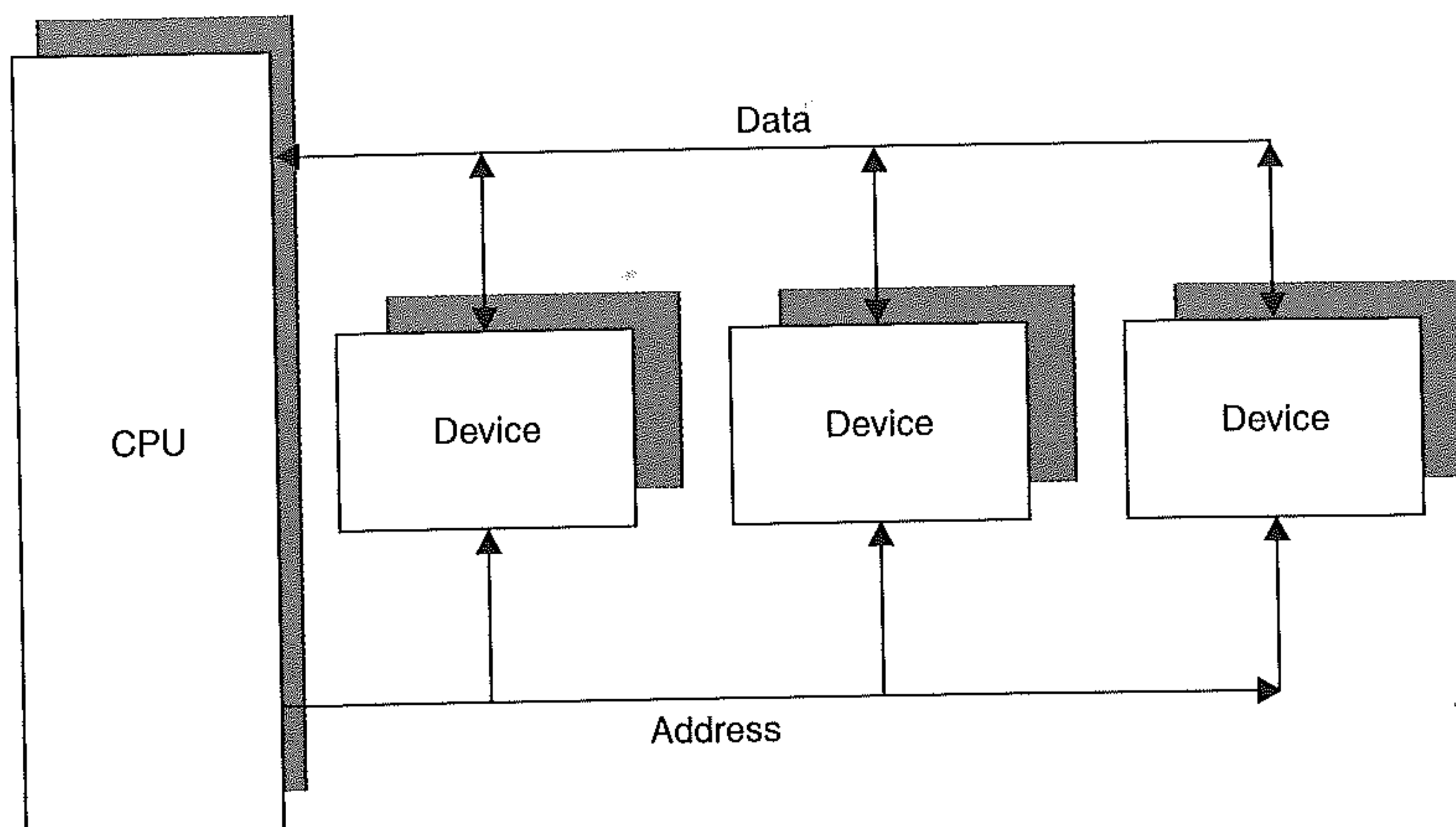


Figure 14.2 Memory-mapped architecture.

assembly instructions: one for accessing memory and one for accessing device registers. The latter is sometimes called **port-mapped I/O** and normally take the form of:

```
IN    AC, PORT
OUT   AC, PORT
```

where **IN** reads the contents of the device register identified by **PORT** into the accumulator **AC**, and **OUT** writes the contents of the accumulator to the device register. (The term **PORT** is used here to indicate an address on the I/O device bus.) There may also be other instructions for reading a device's status. The Intel 486 and the Pentium are examples of an architecture which allows devices to be accessed via special instructions.

With devices on the same logical bus, certain addresses will access a memory location and others will access a device. This approach is called **memory-mapped I/O**. The Motorola M68000 and PowerPC range of computers have memory-mapped I/O.

It is necessary to interface with a device in order to control the device's operations (for example, initializing the device and preparing the device for the transfer of data) and to control the data transfer (for example, initiating or performing the data transfer). It is possible to describe two general mechanisms for performing and controlling input/output; these are status-driven control mechanisms and interrupt-driven control mechanisms.

14.1.1 Status-driven

With this type of control mechanism, a program performs explicit tests in order to determine the status of a given device. Once the status of the device has been determined, the program is able to carry out the appropriate actions. Typically there are three kinds of hardware instruction supporting this type of mechanism. These are:

- test operations that enable the program to determine the status of the given device;
- control operations that direct the device to perform non-transfer device-dependent actions such as positioning disk read heads;
- I/O operations that perform the actual transfer of data between the device and the CPU.

Although historically devices with status-driven I/O were common, as they were inexpensive, nowadays, because of the continuing decrease in hardware cost, most devices are interrupt-driven. However, interrupts can of course be turned off and polling of device status used instead. Interrupts also add to the non-deterministic behaviour of real-time systems and are sometimes prohibited on safety grounds – for example if it is not possible to bound the number of interrupts.

14.1.2 Interrupt-driven

Even with the interrupt-driven mechanism, there are many possible variations depending on how the transfers need to be initiated and controlled. Three of these variations are: interrupt-driven program-controlled, interrupt-driven program-initiated and interrupt-driven channel-program controlled.

Interrupt-driven program-controlled

Here, a device requests an interrupt as a result of encountering certain events, for example the arrival of data. When the request is acknowledged, the processor suspends the executing task and invokes a designated interrupt-handling task which performs appropriate actions in response to the interrupt. When the interrupt-handling task has performed its function, the state of the processor is restored to its state prior to the interrupt, and control of the processor is returned to the suspended task.

I/O controllers – interrupt-driven program-initiated

This type of input/output mechanism is often referred to as an **I/O Controller** or **Direct Memory Access** (or DMA). A DMA device is positioned between the input/output device

and main memory. The DMA device takes the place of the processor in the transfer of data between the I/O device and memory. Although the I/O is initiated by the program, the DMA device controls the actual transfers of data (one block at a time). For each piece of data to be transferred, a request is made by the DMA device for a memory cycle and the transfer is made when the request is granted. When the transfer of the entire block is completed, a *transfer complete* interrupt is generated by the device. This is handled by the interrupt-driven program-controlled mechanism.

The term **cycle stealing** is often used to refer to the impact that DMA devices have on the access to memory. It can lead to non-deterministic behaviour and makes it very difficult to calculate the worst-case execution time of a program (see Section 11.13).

Interrupt-driven channel-program controlled

Channel-program controlled input/output extends the concept of program-initiated input/output by eliminating as much as possible the involvement of the central processor in the handling of input/output devices. The mechanism consists of three major components: the hardware channel and connected devices, the channel program (often called a 'script') and the input/output instructions.

The hardware channel's operations include those of the DMA devices given above. In addition, the channel directs device control operations as instructed by the channel program. The execution of channel programs is initiated from within the application. Once a channel has been instructed to execute a channel program, the selected channel and device proceed independent of the central processor until the specified channel program has been completed or some exceptional condition has occurred. Channel programs normally consist of one or more channel control words which are decoded and executed one at a time by the channel.

In many respects, a hardware channel can be viewed as an autonomous processor sharing memory with the central processor; hence it is often called an I/O coprocessor. Its impact on memory access times of the central processor can be unpredictable (as with DMA devices).

14.1.3 Elements required for interrupt-driven devices

As can be seen from the previous section, the role of interrupts in controlling input/output is an important one. They allow input/output to be performed asynchronously and so avoid the 'busy waiting' or constant status checking that is necessary if a purely status-controlled mechanism is used.

In order to support interrupt-driven input and output, the following mechanisms are required.

Context-switching mechanism

When an interrupt occurs, the current processor state must be preserved and the appropriate service routine activated. Once the interrupt has been serviced, the original task is resumed and execution continues. Alternatively, a new task may be selected by the scheduler as a consequence of the interrupt. This whole task is known as **context**

switching and its actions can be summarized as follows:

- preserving the state of the processor immediately prior to the occurrence of the interrupt;
- placing the processor in the required state for processing the interrupt;
- restoring the suspended task state after the interrupt tasking has been completed.

The state of the task executing on a processor consists of:

- the memory address of the current (or next) instruction in the execution sequence;
- the program status information (this may contain information concerning the mode of processing, current priority, memory protection, allowed interrupts, and so on);
- the contents of the programmable registers.

The type of context switching provided can be characterized by the extent of the task state preservation, and restoration, that is performed by the hardware. Three levels of context switching can be distinguished:

- **basic** – just the program counter is saved and a new one loaded;
- **partial** – the program counter and the program status registers are saved and new ones loaded;
- **complete** – the full context of the task is saved and a new context loaded.

Depending on the degree of processor state preservation required, it may be necessary to supplement the actions of the hardware by explicit software support. For example, a partial context switch may be adequate for an interrupt-handling model which views the handler as a procedure or subroutine. However, if the handler is viewed as being a separate task with its own stack and data areas, there will need to be a low-level handler which does a full context switch between the interrupted task and the handling task. If, on the other hand, the hardware undertakes a full context switch, no such low-level handler is required. Most processors provide only a partial context switch. The ARM processor, for example, supplies a fast interrupt where some of the general-purpose registers are also saved.

It should be noted that some modern processors allow instructions to be fetched, decoded and executed in parallel. Some also allow instructions to be executed out of order from that specified in the program. This chapter will assume that interrupts are **precise** (Walker and Cragon, 1995) in that, when an interrupt handler executes:

- all instructions issued before the interrupted instruction have finished executing and have modified the program state correctly;
- instructions issued after the interrupted instruction have not been executed and no program state has been modified;
- if the interrupted instruction itself caused the interrupt (for example, an instruction which causes memory violation) then either the instruction has been completely executed or not executed at all.

If an interrupt is not precise, it is assumed that the recovery is transparent to the interrupt-handling software. See Walker and Cragon (1995) for a taxonomy of interrupt handling.

Interrupting device identification

Devices differ in the way they must be controlled; consequently, they will require different interrupt-handling routines. In order to invoke the appropriate handler, some means of identifying the interrupting device must exist. Four interrupting device identification mechanisms can be distinguished: vectored, status, polling and high-level language primitive.

A **vectored** mechanism for identifying the interrupting device consists of a set of dedicated (and usually contiguous) memory locations called an **interrupt vector table** and a hardware mapping of device addresses onto the interrupt vector. An interrupt vector may be used by one particular device or may be shared by several devices. The programmer must associate a particular interrupt vector location explicitly with an interrupt-service routine. This may be done either by setting the vector location to the address of the service routine, or by setting it to an instruction that causes a branch to occur to the required routine. In this way, the service routine is directly tied to an interrupt vector location, which in turn is indirectly tied to a device or set of devices. Therefore, when a particular service routine is invoked and executed, the interrupting device has been implicitly identified.

A **status** mechanism for identifying an interrupting device is used for machine configurations on which several devices are connected to a device controller and they do not have unique interrupt vectors. It is also used in the case where a generalized service routine will initially handle all interrupts. With this mechanism, each interrupt has an associated status word which specifies the device causing the interrupt and the reason for the interrupt (among other things). The status information may be provided automatically by the hardware in a dedicated memory location for a given interrupt or class of interrupts, or it may need to be retrieved by means of some appropriate instruction.

The **polling** device identification mechanism is the simplest of all. When an interrupt occurs, a general interrupt-service routine is invoked to handle the interrupt. The status of each device is interrogated in order to determine which device has requested the interrupt. When the interrupting device has been identified, the interrupt is serviced in the appropriate manner.

With some modern computer architectures, interrupt handling is directly associated with a **high-level language primitive**. With these systems, an interrupt is often viewed as a synchronization message down an associated channel. In this case, the device is identified by the channel which becomes active.

Interrupt identification

Once the device has been identified, the appropriate interrupt-handling routine must determine why it generated the interrupt. In general, this can be supplied either by status information provided by the device or by having different interrupts from the same device occurring through different vectored locations or channels.

Interrupt control

Once a device is switched on and has been initialized, although it may be able to produce interrupts, they will be ignored unless the device has had its interrupts enabled. This control (enabling/disabling) of interrupts may be performed by means of the following interrupt control mechanisms.

- **Status interrupt control** mechanisms provide flags, either in an interrupt state table or via device and program status words, to enable and disable the interrupts. The flags are accessible (and may be modified) by normal bit-oriented instructions or special bit-testing instructions.
- **Mask interrupt control** mechanisms associate each device interrupt with a particular bit position in a word. If the bit is set to one the interrupt will be blocked; if it is set to zero then it will be allowed. The interrupt mask word may be addressable by normal bit-oriented (or word-oriented) instructions or may be accessible only through special interrupt-masking instructions.
- **Level interrupt control** mechanisms have devices associated with certain levels. The current level of the processor determines which devices may or may not interrupt. Only those devices with a higher logical level may interrupt. When the highest logical level is active, only those interrupts which cannot be disabled (for example, power fail) are allowed. This does not explicitly disable interrupts, so interrupts at a lower logical level than the current processor level will still be generated, and will not need to be re-enabled when the processor level falls appropriately.

Priority control

Some devices have higher urgency than others, and therefore a priority facility is often associated with interrupts. This mechanism may be static or dynamic and is usually related to the device interrupt control facility and the priority levels of the processor.

14.1.4 A simple example I/O system

In order to illustrate the various components of an I/O system, a simple machine is described. It is loosely based on the Motorola 68000 series of computers.

Each device supported on the machine has as many different types of register as are necessary for its operation. These registers are memory-mapped. The most common types used are **control and status** registers which contain all the information on a device's status, and allow the device's interrupts to be enabled and disabled. **Data buffer** registers act as buffer registers for temporarily storing data to be transferred into or out of the machine via the device.

A typical control and status register for the computer has the following structure:

bits		
15 - 12	: Errors	-- set when device errors occur
11	: Busy	-- set when the device is busy

10 - 8	: Unit select	-- where more than one device is -- being controlled
7	: Done/ready	-- I/O completed or device ready
6	: Interrupt enable	-- when set enables interrupts
5 - 3	: reserved	-- reserved for future use
2 - 1	: Device function	-- set to indicate required function
0	: Device enable	-- set to enable the device

The typical structure of a data buffer register used for a character-orientated device is:

```
bits
15 - 8   : Unused
7 - 0    : Data
```

Bit 0 is the least significant bit of the register.

A device may have more than one of each of these registers, the exact number being dependent on the nature of the device. For example, one particular Motorola parallel interface and timer device has 14 registers.

Interrupts allow devices to notify the processor when they require service; they are vectored. When an interrupt occurs, the processor stores the current program counter (PC) and the current program status word (PSW) on the system stack. The PSW will contain, among other things, the processor priority. Its actual layout is given below:

```
bits
15 - 11   : Mode information
10 - 8     : Unused
7 - 5      : Priority
4 - 0      : Condition codes
```

The condition codes contain information on the result of the last processor operation.

The new PC and PSW are loaded from two preassigned consecutive memory locations (the interrupt vector). The first word contains the address of the interrupt service routine and the second contains the PSW, including the priority at which the interrupt is to be handled. A low-priority interrupt handler can be interrupted by a higher-priority interrupt.

This example I/O system will be returned to later.

14.2 Language requirements

As noted above, one of the main characteristics of an embedded system is the need to interact with input and output devices, all of which have their particular characteristics. The programming of such devices has traditionally been the stronghold of the assembly language programmer, but languages like C, Java and Ada have now attempted to provide progressively higher-level mechanisms for these low-level functions. This makes device-driving and interrupt-handling routines easier to read, write and maintain. The support can be partitioned into two components: **modularity and encapsulation facilities** and an **abstract model** of device handling.

14.2.1 Modularity and encapsulation facilities

Low-level device interfacing is necessarily machine-dependent, and therefore in general it is not portable.¹

In any software system, it is important to separate the non-portable sections of code from the portable ones. Wherever possible, it is advisable to encapsulate all the machine-dependent code into units which are clearly identifiable. In Ada, the package and protected type facilities are used. In Java, classes and packages are the appropriate encapsulation mechanisms. In C, the only facility is a file.

14.2.2 Abstract models of device handling

A device can be considered to be a processor performing a fixed task. A computer system can, therefore, be considered to be a collection of parallel tasks. There are several models by which the device 'task' can communicate and synchronize with the tasks executing inside the main processor. All models must provide:

- (1) **Facilities for representing, addressing and manipulating device registers** – a device register may be represented as a program variable, an object or even a communication channel.
- (2) **A suitable representation of interrupts** – the following representations are possible:
 - (a) **Procedure call** – the interrupt is viewed as a procedure call (in a sense, it is a remote procedure call coming from the device task). Any communication and synchronization required must be programmed in the interrupt handler procedure. The procedure is non-nested: only global state or state local to the handler can be accessed.
 - (b) **Sporadic task invocation** – the interrupt is viewed as a request to execute a task. The handler is a sporadic task and it can access both local persistent data and global data (if shared-variable communication is available in the concurrency model).
 - (c) **Asynchronous notification** – the interrupt is viewed as an asynchronous notification directed at a task. The handler can access both the local state of the task and the global state. Both the resumption model and the termination are possible.
 - (d) **Shared-variable condition synchronization** – the interrupt is viewed as a condition synchronization within a shared-variable synchronization mechanism; for example, a signal operation on a semaphore or a send operation on a condition variable in a monitor. The handler can access both the local state of the task/monitor and the global state.

¹Note that in recent years there has been some attempt to produce a Uniform Driver Interface (UDI). UDI defines an architecture and a set of operating-system neutral (and hardware neutral) interfaces for use between the device driver and the surrounding system. This allows device drivers and operating systems to be developed independently, and multiple operating systems and hardware platforms to use the same driver.

- (e) **Message-based synchronization** – the interrupt is viewed as a content-less message sent down a communication channel. The receiving task can access the local state of the task.

All of the above approaches, except the procedural approach, require a full context switch as the handler executes in the scope of a task. Optimizations are possible if the handlers are restricted. For example, if the handler in the asynchronous notification model has resumption semantics and it does not access any data local to the task, then the interrupt can be handled with only a partial context switch.

Not all these models can be found in real-time languages and operating systems. The most popular one is the procedural model as this requires little support. Usually, real-time systems implemented in C and C++ adopt this model with device registers represented as program variables. For sequential systems, the asynchronous event model is identical in effect to the procedural model, as there is only one task to interrupt and therefore there is no need to identify the task or the event. The Ada model is a hybrid between a procedure model and a shared-variable condition synchronization model. The interrupt is mapped to a protected procedure call and registers are accessed via program variables. Real-Time Java views an interrupt as the firing of an asynchronous event where the handler is a schedulable object. Older languages supported other models. For example, Modula-1 and Real-Time Euclid mapped interrupts to signals on condition variables and semaphores respectively (again registers are represented as program variables) and are, therefore, pure shared-variable models. Occam2 viewed an interrupt as a message down a channel with device registers also represented as channels.

The next three sections now consider Ada, Real-Time Java and C in detail.

14.3 Ada

In Ada, there are three ways in which tasks can synchronize and communicate with each other:

- through the rendezvous
- using protected units
- via shared variables.

In general, Ada assumes that the device and the program have access to shared memory device registers which can be specified using its representation specification techniques. In Ada 83, interrupts were represented by hardware-generated task entry calls. In the current version of Ada, this facility is considered obsolete. Consequently, it will not be discussed in this book.

The preferred method of device driving is to encapsulate the device operations in a protected unit. An interrupt is mapped to a protected procedure call.

14.3.1 Addressing and manipulating device registers

Ada presents the programmer with a comprehensive set of facilities for specifying the implementation of data types. These are collectively known as **representation aspects**,

and they indicate how the types of the language are to be mapped onto the underlying hardware. A type can only have a single representation. The representation is specified separately from the logical structure of the type. Of course, the specification of the representation of a type is optional and can be left to the compiler.

Representation aspects are a compromise between abstract and concrete structures. Four distinct specifications are available.

- (1) **Attribute definition clause** – allows various attributes of an object, task or sub-program to be set; for example, the size (in bits) of objects, the storage alignment, the maximum storage space for a task, the address of an object.
- (2) **Enumeration representation clause** – the literals of an enumeration type may be given specific internal values.
- (3) **Record representation clause** – record components can be assigned offsets and lengths within single storage units.
- (4) **At clause** – this was the main Ada 83 mechanism for positioning an object at a specific address; this facility is now obsolete (attributes can be used) and will not, therefore, be discussed further.

If an implementation cannot obey a specification request then the compiler must either reject the program or raise an exception at run-time.

In order to illustrate the use of these mechanisms, consider the following type declarations which represent a typical control and status register of the simple machine defined earlier.

```

type Error_T is (None, Read_Error, Write_Error,
                  Power_Fail, Other);
type Function_T is (Read, Write, Seek);
type Unit_T is new Integer range 0 .. 7;

type Csr_T is record
  Errors      : Error_T;
  Busy        : Boolean;
  Unit        : Unit_T;
  Done        : Boolean;
  Ienable     : Boolean;
  Dfun        : Function_T;
  Denable     : Boolean;
end record;

```

An enumeration representation clause specifies the internal codes for the literals of the enumeration type. For example, the internal codes for the function required by the device above may be:

```

01  --  READ
10  --  WRITE
11  --  SEEK

```

In Ada, this is specified by:

```

type Function_T is (Read, Write, Seek);
for Function_T use (Read=> 1, Write => 2, Seek => 3);

```

Similarly, for `Error_t`:

```
type Error_T is (None, Read_Error, Write_Error, Power_Fail, Other);
for Error_T use (None => 0, Read_Error => 1, Write_Error => 2,
                 Power_Fail => 3, Other => 4);
-- note, this is in fact the default assignment
```

A record representation clause specifies the storage representation of records; that is, the order, position and size of its components. The bits in the record are numbered from 0; the range in the component clause specifies the number of bits to be allocated.

For example, the control status register is given by:

```
Word : constant := 2; -- number of storage units in a word
Bits_In_Word : constant := 16; -- bits in word
for Csr_T use record
  Denable   at 0*Word range 0..0; -- at word 0 bit 0
  Dfun      at 0*Word range 1..2;
  Ienable   at 0*Word range 6..6;
  Done      at 0*Word range 7..7;
  Unit      at 0*Word range 8 .. 10;
  Busy      at 0*Word range 11 .. 11;
  Errors    at 0*Word range 12 .. 15;
end record;

for Csr_T'Size use Bits_In_Word; -- the size of object of Csr type
for Csr_T'Alignment use Word; -- object should be word aligned
for Csr_T'Bit_order use Low_Order_First;
-- first bit is least significant bit of byte
```

A size attribute specifies the amount of storage that is to be associated with a type. In this example, the register is a single 16-bit word. The alignment attribute specifies that the compiler should always place objects on an integral number of storage units boundary, in this case a word boundary. The bit-ordering attribute specifies whether the machine numbers the most significant bit as 0 (big endian) or the least significant bit (little endian). Note that bits 3, 4 and 5 (which were reserved for future use) have not been specified.

Finally, an actual register needs to be declared and placed at the required location in memory. In Ada, `Address` is an implementation-defined type defined in package `System`. A child package (`System.Storage_Elements.To_Address`) provides a function for converting an integer value into the address type.

```
Tcsr : Csr_T;
for Tcsr'Address use System.Storage_Elements.To_Address(8#177566#);
```

Having now constructed the abstract and concrete data representation of the register, and placed an appropriately defined variable at the correct address, the hardware register can be manipulated by assignments to this variable:

```
Tcsr := (Denable => True, Dfun => Read,
         Ienable => True, Done => False,
         Unit => 4, Errors => None);
```


The use of this record aggregate assumes that the entire register will be assigned values at the same time. To ensure that `Dfun` is not set before the other fields of the record it may be necessary to use a temporary (shadow) control register:

```
Temp_Cr : Csr_T;
```

This temporary register is then assigned control values and copied into the real register variable:

```
Tcsr := Temp_Cr;
```

The code for this assignment will in most cases ensure that the entire control register is updated in a single action. If any doubt still remains, then the pragma `Atomic` can be used (which instructs the compiler to generate the update as a simple operation or produce an error message).

After the completion of the I/O operation, the device itself may alter the values on the register; this is recognized in the program as changes in the values of the record components:

```
if Tcsr.Errors = Read_Error then
  raise Disk_Error;
end if;
```

The object `Tcsr` is therefore a collection of shared variables, which are shared between the device control task and the device itself. Mutual exclusion between these two concurrent (and parallel) tasks is necessary to give reliability and performance. This is achieved in Ada by using a protected object.

14.3.2 Interrupt handling

Ada defines the following model of an interrupt.

- An interrupt represents a class of events that are detected by the hardware or the system software.
- The **occurrence** of an interrupt consists of its **generation** and its **delivery**.
- The generation of an interrupt is the event in the underlying hardware or system which makes the interrupt available to the program.
- Delivery is the action which invokes a part of the program (called the interrupt handler) in response to the interrupt occurrence. In between the generation of the interrupt and its delivery, the interrupt is said to be **pending**. The handler is invoked once for each delivery of the interrupt. The **latency** of an interrupt is the time spent while in the pending state.
- While an interrupt is being handled, further interrupts from the same source are **blocked**; all future occurrences of the interrupt are prevented from being delivered. It is usually device-dependent as to whether a blocked interrupt remains pending or is lost.

- Certain interrupts are **reserved**. The programmer is not allowed to provide a handler for a reserved interrupt. Usually, a reserved interrupt is handled directly by the run-time support system of Ada (for example, a clock interrupt used to implement the delay statement).
- Each non-reserved interrupt has a default handler that is assigned by the run-time support system.

Handling interrupts using protected procedures

The main representation of an interrupt handler in Ada is a parameterless protected procedure. Each interrupt has a unique discrete identifier which is supported by the system. How this unique identifier is represented is implementation-defined; it might, for example, be the address of the hardware interrupt vector associated with the interrupt. Where Ada is implemented on top of an operating system that supports signals, each signal may be mapped to a particular interrupt identifier, thus allowing signal handlers to be programmed in the language.

Identifying interrupt-handling protected procedures is done using one of two pragmas:

```
pragma Attach_Handler(Handler_Name, Expression);
-- This can appear in the specification or body of a
-- library-level protected object and allows the
-- static association of a named handler with the
-- interrupt identified by the expression; the handler
-- becomes attached when the protected object is created.
-- Raises Program_Error:
--   (a) when the protected object is created and
--       the interrupt is reserved,
--   (b) if the interrupt already has a
--       user-defined handler, or
--   (c) if any ceiling priority defined is
--       not in the range Ada.Interrupt_Priority.

pragma Interrupt_Handler(Handler_Name);
-- This can appear in the specification of a library-level
-- protected object and allows the dynamic association of
-- the named parameterless procedure as an interrupt
-- handler for one or more interrupts. Objects created
-- from such a protected type must be library level.
```

Program 14.1 defines the Systems Programming Annex's support for interrupt identification and the dynamic attachment of handlers. In all cases where `Program_Error` is raised, the currently attached handler is not changed.

It should be noted that the Reference function provides the mechanisms by which interrupt task entries are supported. As mentioned earlier, this model of interrupt handling is considered obsolete and should, therefore, not be used.

It is possible that an implementation will also allow the association of names with interrupts via Program 14.2. This will be used in the following example.

Program 14.1 Package Ada.Interrupts.

```
package Ada.Interrupts is
  type Interrupt_Id is implementation_defined; -- must be discrete
  type Parameterless_Handler is access protected procedure;

  function Is_Reserved(Interrupt : Interrupt_Id) return Boolean;
    -- Returns True if the interrupt is reserved,
    -- returns False otherwise.
  function Is_Attached(Interrupt : Interrupt_Id) return Boolean;
    -- Returns True if the interrupt is attached to a
    -- handler, returns False otherwise.
    -- Raises Program_Error if the interrupt is reserved.
  function Current_Handler(Interrupt : Interrupt_Id)
    return Parameterless_Handler;
    -- Returns an access variable to the current handler for
    -- the interrupt. If no user handler has been attached, a
    -- value is returned which represents the default handler.
    -- Raises Program_Error if the interrupt is reserved.
  procedure Attach_Handler(New_Handler : Parameterless_Handler;
    Interrupt : Interrupt_Id);
    -- Assigns New_Handler as the current handler.
    -- If New_Handler is null, the default handler is restored.
    -- Raises Program_Error:
    --   (a) if the protected object associated with the
    --       New_Handler has not been identified with a
    --       pragma Interrupt_Handler,
    --   (b) if the interrupt is reserved,
    --   (c) if the current handler was attached statically
    --       using pragma Attach_Handler.
  procedure Exchange_Handler(
    Old_Handler : out Parameterless_Handler;
    New_Handler : Parameterless_Handler;
    Interrupt : Interrupt_Id);
    -- Assigns New_Handler as the current handler for the
    -- Interrupt and returns the previous handler in
    -- Old_Handler.
    -- If New_Handler is null, the default handler is restored.
    -- Raises Program_Error:
    --   (a) if the protected object associated with the
    --       New_Handler has not been identified with a
    --       pragma Interrupt_Handler,
    --   (b) if the interrupt is reserved,
    --   (c) if the current handler was attached statically
    --       using pragma Attach_Handler.
  procedure Detach_Handler(Interrupt : Interrupt_Id);
    -- Restores the default handler for the specified interrupt.
    -- Raises Program_Error:
    --   (a) if the interrupt is reserved,
    --   (b) if the current handler was attached statically
    --       using pragma Attach_Handler.
  function Reference(Interrupt : Interrupt_Id)
    return System.Address;
    -- Returns an Address which can be used to attach
    -- a task entry to an interrupt via an address
    -- clause on an entry.
    -- Raises Program_Error if the interrupt cannot be
    -- attached in this way.

private
  ... -- not specified by the language
end Ada.Interrupts;
```

Program 14.2 Package Ada.Interrupts.Names.

```

package Ada.Interrupts.Names is
  implementation_defined : constant Interrupt_Id :=
                                implementation_defined;

  ...
  implementation_defined : constant Interrupt_Id :=
                                implementation_defined;

private
  ... -- not specified by the language
end Ada.Interrupts.Names;

```

14.3.3 A simple driver example

A common class of equipment to be attached to an embedded computer system is the analog-to-digital converter (ADC). The converter samples some environmental factors, such as temperature or pressure; it translates the measurements it receives, which are usually in millivolts, and provides scaled integer values on a hardware register. Consider a single converter that has a 16-bit result register at hardware address 8#150000# and a control register at 8#150002#. The computer is a 16-bit machine and the control register is structured as follows:

Bit	Name	Meaning
0	A/D Start	Set to 1 to start a conversion.
6	Interrupt Enable/Disable	Set to 1 to enable interrupts
7	Done	Set to 1 when conversion is complete.
8-13	Channel	The converter has 64 analog inputs, the particular one required is indicated by the value of the channel.
15	Error	Set to 1 by the converter if device malfunctions.

The driver for this ADC will be structured as a protected type within a package, so that the interrupt it generates can be processed as a protected procedure call, and so that more than one ADC can be catered for:

```

package Adc_Device_Driver is
  Max_Measure : constant := (2**16)-1;
  type Channel is range 0..63;
  subtype Measurement is Integer range 0..Max_Measure;
  procedure Read(Ch: Channel; M : out Measurement);
    -- potentially blocking
  Conversion_Error : exception;
private
  for Channel'Size use 6;
  -- indicates that six bits only must be used
end Adc_Device_Driver;

```


For any request, the driver will make three attempts before raising the exception. The package body follows:

```

with Ada.Interrupts.Names; use Ada.Interrupts;
with System; use System;
with System.Storage_Elements; use System.Storage_Elements;
package body Adc_Device_Driver is
  Bits_In_Word : constant := 16;
  Word : constant := 2; -- bytes in word
  type Flag is (Down, Set);

  type Control_Register is
    record
      Ad_Start : Flag;
      Ienable  : Flag;
      Done     : Flag;
      Ch       : Channel;
      Error    : Flag;
    end record;

  for Control_Register use
    -- specifies the layout of the control register
    record
      Ad_Start at 0*Word range 0..0;
      Ienable  at 0*Word range 6..6;
      Done     at 0*Word range 7..7;
      Ch       at 0*Word range 8..13;
      Error    at 0*Word range 15..15;
    end record;

  for Control_Register'Size use Bits_In_Word;
  -- the register is 16-bits long
  for Control_Register'Alignment use Word;
  -- on a word boundary
  for Control_Register'Bit_order use Low_Order_First;

  type Data_Register is range 0 .. Max_Measure;
  for Data_Register'Size use Bits_In_Word;
  -- the register is 16-bits long

  Contr_Reg_Addr : constant Address := To_Address(8#150002#);
  Data_Reg_Addr  : constant Address := To_Address(8#150000#);
  Adc_Priority   : constant Interrupt_Priority := 63;
  Control_Reg    : aliased Control_Register;
  -- aliased indicates that pointers are used to access it
  for Control_Reg'Address use Contr_Reg_Addr;
  -- specifies the address of the control register
  Data_Reg : aliased Data_Register;
  for Data_Reg'Address use Data_Reg_Addr;
  -- specifies the address of the data register

  protected type Interrupt_Interface(Int_Id : Interrupt_Id;
    Cr : access Control_Register;
    Dr : access Data_Register) is
    entry Read(Chan : Channel; M : out Measurement);

```

```

private
  entry Done(Chan : Channel; M : out Measurement);
  procedure Handler;
  pragma Attach_Handler(Handler, Int_Id);
  pragma Interrupt_Priority(Adc_Priority);
  -- see Section 11.9 for discussion on priorities
  Interrupt_Occurred : Boolean := False;
  Next_Request : Boolean := True;
end Interrupt_Interface;

Adc_Interface : Interrupt_Interface(Names.Adc,
                                     Control_Reg'Access,
                                     Data_Reg'Access);
-- this assumes that 'Adc' is registered as an
-- Interrupt_Id in Ada.Interrupts.Names
-- 'Access gives the address of the object

protected body Interrupt_Interface is

  entry Read(Chan : Channel; M : out Measurement)
    when Next_Request is
    Shadow_Register : Control_Register;
  begin
    Shadow_Register := (Ad_Start => Set, Ienable => Set,
                       Done => Down, Ch   => Chan, Error => Down);
    Cr.all := Shadow_Register;
    Interrupt_Occurred := False;
    Next_Request := False;
    requeue Done;
  end Read;

  procedure Handler is
  begin
    Interrupt_Occurred := True;
  end Handler;

  entry Done(Chan : Channel; M : out Measurement)
    when Interrupt_Occurred is
  begin
    Next_Request := True;
    if Cr.Done = Set and Cr.Error = Down then
      M := Measurement(Dr.all);
    else
      raise Conversion_Error;
    end if;
  end Done;
end Interrupt_Interface;

procedure Read(Ch : Channel; M : out Measurement) is
begin
  for I in 1..3 loop
    begin
      Adc_Interface.Read(Ch,M);
    return;
  end loop;
end Read;

```



```

    exception
        when Conversion_Error => null;
    end;
end loop;
raise Conversion_Error;
end Read;
end Adc_Device_Driver;

```

The client tasks simply call the `Read` procedure indicating the channel number from which to read, and an output variable for the actual value read. Inside the procedure, an inner loop attempts three conversions by calling the `Read` entry in the protected object associated with the converter. Inside this entry, the control register, `Cr`, is set up with appropriate values. Once the control register has been assigned, the client task is requeued on a private entry to await the interrupt. The `Next_Request` flag is used to ensure only one call to `Read` is outstanding.

Once the interrupt has arrived (as a parameterless protected procedure call), the barrier on the `Done` entry is set to `True`; this results in the `Done` entry being executed (as part of the interrupt handler), which ensures that `Cr.Done` has been set and that the error flag has not been raised. If this is the case, the out parameter `M` is constructed, using a type conversion, from the value on the buffer register. (Note that this value cannot be out of range for the subtype `Measurement`.) If the conversion has not been successful, the exception `Conversion_Error` is raised; this is trapped by the `Read` procedure, which makes three attempts in total at a conversion before allowing the exception to propagate.

The above example illustrates that it is often necessary when writing device drivers to convert objects from one type to another. In these circumstances the strong typing features of Ada can be an irritant. It is, however, possible to circumvent this difficulty by using a generic function that is provided as a predefined library unit:

```

generic
    type Source (<>) is limited private;
    type Target (<>) is limited private;
function Ada.Unchecked_Conversion(S : Source) return Target;
pragma Convention(Intrinsic, Ada.Unchecked_Conversion);
pragma Pure(Ada.Unchecked_Conversion);

```

The effect of unchecked conversion is to copy the bit pattern of the source over to the target. The programmer must make sure that the conversion is sensible and that all possible patterns are acceptable to the target.

14.3.4 Accessing I/O devices through special instructions

If special instructions are required, assembler code may have to be integrated with Ada code. The machine code insertion mechanism enables programmers to write Ada code which contains visible non-Ada objects. This is achieved in a controlled way by only allowing machine code instructions to operate within the context of a subprogram body. Moreover, if a subprogram contains code statements then it can contain only code statements and 'use' clauses (comments and pragmas being allowed as usual).

As would be expected, the details and characteristics of using code inserts are largely implementation dependent; implementation-specific pragmas and attributes may be used to impose particular restrictions and calling conventions on the use of objects defining code instructions. A code statement has the following structure:

```
code_statement ::= qualified_expression
```

The qualified expression should be of a type declared within a predefined library package called `System.Machine_Code`. It is this package that provides record declarations (in standard Ada) to represent the instructions of the target machine. The following example illustrates the approach:

```
D : Data; -- to be input

procedure In_Op; pragma Inline(In_Op);

procedure In_Op is
  use System.Machine_Code;
begin
  My_Machine_Format'(Code => In_Instruction, Reg => 1, Port => 1);
  My_Machine_Format'(CODE => SAVE, REG =>, S'Address);
end;
```

The pragma `Inline` instructs the compiler to include inline code, rather than a procedure call, whenever the subprogram is used.

Even though this code insertion method is defined in Ada, the language makes it quite clear (ARM 13.8.4) that an implementation need not provide a `Machine_Code` package unless the Systems Programming Annex is supported. If it does not, the use of machine code inserts is prohibited.

14.4 Real-Time Java

Although Java was initially designed for embedded system and is awash with facilities for graphics programming and file handling, the language says nothing about how to program device drivers or handle interrupts. Real-Time Java attempts to correct this situation, but currently provides only limited support.

14.4.1 Addressing and manipulating device registers

Real-Time Java supports the ability to access device registers through the notion of raw memory. An implementation is allowed to support a range of memory types such as DMA memory or shared memory. One such memory could be the memory where I/O registers are allocated: `IO_Page`. The `RawMemoryAccess` class (defined in Program 14.3) can then be used to read and write to that memory. The approach does not allow user-defined objects to be mapped to raw memory, as this could potentially allow violations of the typing mechanism. Consequently, it is necessary to access individual registers as primitive data types (byte, int, short, long) and use low-level bit-wise operations.

Program 14.3 An excerpt of the RawMemoryAccess class.

```
public class RawMemoryAccess {
    protected RawMemoryAccess(long base, long size);
    protected RawMemoryAccess(RawMemoryAccess memory, long base,
                               long size);

    public static RawMemoryAccess create(java.lang.Object type,
                                          long size)
        throws SecurityException, OffsetOutOfBoundsException,
               SizeOutOfBoundsException,
               UnsupportedPhysicalMemoryException;

    public static RawMemoryAccess create(java.lang.Object type,
                                          long base, long size)
        throws SecurityException, OffsetOutOfBoundsException,
               SizeOutOfBoundsException,
               UnsupportedPhysicalMemoryException;

    public byte getByte(long offset)
        throws SizeOutOfBoundsException, OffsetOutOfBoundsException;

    public void getBytes(long offset, byte[] bytes, int low,
                        int number) throws
        SizeOutOfBoundsException, OffsetOutOfBoundsException;

    // similarly for integers, long integers and short integers

    public void setByte(long offset, byte value) throws
        SizeOutOfBoundsException, OffsetOutOfBoundsException;

    public void setBytes(long offset, byte[] bytes, int low,
                        int number) throws
        SizeOutOfBoundsException, OffsetOutOfBoundsException;

    // similarly for integers, long integers and short integers
    . . .
}
```

Consider again the control and status register for the simple ADC given in Section 14.3.3.

Bit	Name	Meaning
0	A/D Start	Set to 1 to start a conversion.
6	Interrupt Enable/Disable	Set to 1 to enable interrupts
7	Done	Set to 1 when conversion is complete.
8-13	Channel	The converter has 64 analog inputs, the particular one required is indicated by the value of the channel.
15	Error	Set to 1 by the converter if device malfunctions.

First it is necessary to create a class for the register. The constructor for this class creates an instance of the `RawMemoryAccess` class indicating the memory type as being the `IO_Page`. The `setControlWord` method can then be used to access the register itself.

```
public class ControlAndStatusRegister {
    RawMemoryAccess rawMemory;

    public ControlAndStatusRegister(long base, long size)
    {
        rawMemory = RawMemoryAccess.create(IO_Page, base, size);
    }

    public void setControlWord(short value)
    {
        rawMemory.setShort(0, value);
    }
}
```

Now, using a shadow device register and bit-wise logical operators, the correct bit pattern can be constructed. For example, to start a conversion on channel 6:

```
short shadow, channel;
final short start = 01;
final short enable = 040;
final long csrAddress = 015002;
final long csrSize = 2;
ControlAndStatusRegister csr = new
    ControlAndStatusRegister(csrAddress, csrSize);

channel = 6;
shadow = (channel << 8) | start | enable);
csr.setControlWord(shadow);
```

14.4.2 Interrupt handling

Real-Time Java views an interrupt as an asynchronous event (see Section 7.7.1). An interrupt occurring is thus equivalent to the `fire` method being called. The association between the interrupt and the event is achieved by calling the `bindTo` method in the `AsyncEvent` class. The parameter is of string type, and this is used in an implementation-dependent manner – one approach might be to pass the address of the interrupt vector. When the interrupt occurs, the appropriate asynchronous event's `fire` method is called. Now, it is possible to associate the handler with a schedulable object and give it appropriate priority and release parameters.

```
AsyncEvent Interrupt = new AsyncEvent();
AsyncEventHandler InterruptHandler = new BoundAsyncEventHandler(
    priParams, releaseParams, null, null, null);

Interrupt.addHandler(InterruptHandler);
Interrupt.bindTo("177760");
```

Program 14.4 An excerpt of the POSIXSignalHandler class.

```

public final class POSIXSignalHandler {
    public static final int SIGABRT;
    public static final int SIGALRM;
    public static final int SIGBUS
    . . .

    public static synchronized void addHandler(int signal,
        AsyncEventHandler handler);

    public static synchronized void removeHandler(int signal,
        AsyncEventHandler handler);

    public static synchronized void setHandler(int signal,
        AsyncEventHandler handler);
}

```

For the case where the Real-Time Java program is executing on top of a POSIX-compliant operating system, the class `POSIXSignalHandler` (see Program 14.4) can be used to associate asynchronous event handlers with the occurrence of a POSIX signal. Interestingly, POSIX's real-time signals are not supported.

14.5 C and older real-time languages

The first generation of real-time programming languages (RTL/2, Coral 66 and so on) provide no real support for concurrent programming or for the programming of devices. Interrupts are typically viewed as procedure calls, and very often the only facility available for accessing device registers is to allow assembly language code to be embedded in the program.

Another common feature with early real-time languages is that they tend to be weakly typed. Therefore variables can be treated as fixed-length bit strings. This allows the individual bits and fields of registers to be manipulated using low-level operators, such as logical shift and rotate instructions. However, the disadvantages of having weak typing by far outweigh the benefits of this flexibility.

The language C has continued with this tradition. Device registers are addressed by pointer variables which can be assigned to the memory location of the register. They are manipulated either by low-level bit-wise logical operators or by the use of bitfields in structure definitions. The latter appears to be similar to record representation clauses in Ada, but in fact is both machine- and compiler-dependent. To illustrate the use of the bit manipulation facilities of C, two examples are presented, the first using low-level bit-wise logical operators and the second using bitfields.

Consider again the control and status register for the simple ADC given in Section 14.3.3. Using bit-wise logical operators, it is first necessary to define a set of masks corresponding to each bit position.

```

#define START    01 /* numbers beginning with 0 are hexadecimal */
#define ENABLE   040
#define ERROR    0100000

```

The 'Channel' field can either be defined on a per bit basis, or the value can be shifted to its correct position. The latter approach is used below, where the channel number 6 is required:

```

unsigned short int *register, shadow, channel;

register = 0xAA12;
channel = 6;
shadow = 0;

shadow |= (channel << 8) | START | ENABLE
*register = shadow

```

With bitfields, this becomes:

```

struct {
    unsigned int start      : 1; // field 1-bit long
    unsigned int            : 5; // unnamed field 5-bits long
    unsigned int interrupt  : 1; // field 1-bit long
    unsigned int Done       : 1; // field 1-bit long
    unsigned int Channel    : 6; // field 6-bits long
    unsigned int error      : 1; // field 1-bit long
} control_register;

control_register *register, shadow;
register = 0xAA12;
shadow.start = 1;
shadow.Interrupt = 1;
shadow.channel = 6;
shadow.error = 0;

*register = shadow;

```

There are two points to note about this example.

- C gives no guarantee of the ordering of fields; hence the compiler may decide to pack the fields in the word in a different order from that implied by the programmer.
- C does not attempt to address whether the machine numbers bits from left to right or right to left.

Given these two points, it is clear that bitfields should not be used to access device registers unless the programmer has knowledge of how they are implemented by the particular compiler for the specific machine being used. Even in this case, the code will not be portable.

For portability, the C programmer is therefore forced to use low-level bit-wise logical operators. The following example shows how these can quickly become unreadable

(although arguably they produce more efficient code). The following procedure sets *n* bits starting at position *p* in the register pointed at by *reg* to *x*;

```
unsigned int setbits(unsigned int *reg, unsigned int n,
                    unsigned int p, unsigned int x)
{
    unsigned int data, mask;

    data = (x & ~(~0 << n)) << (p); /* data to be masked in */
    mask = ~(~0 << n); /* mask */
    *reg &= ~(mask << (p)); /* clear current bits */
    *reg |= data; /* OR in the data */
}
```

The C code is somewhat terse in this example: `~` means a bit-wise complement, `<<` is a shift left (with a 0 fill), `&` is a bit-wise ‘and’, and `|` is a bit-wise ‘or’.

With the simple I/O architecture outlined in Section 14.1.4, interrupt handlers are assigned by placing the address of a parameterless procedure in the appropriate interrupt vector location. Once the procedure is executed, any communication and synchronization with the rest of the program must be programmed directly.

Although POSIX provides alternative mechanisms which, in theory, could be used to provide an alternative model of interrupt handling (for example, associating an interrupt with a condition variable), there is currently no standard mechanism for attaching user-defined handlers to interrupts.

14.6 Scheduling device drivers

As many real-time systems have I/O components, it is important that the scheduling analysis incorporates any features which are particular to this low-level programming. It has already been noted that DMA and channel-program controlled techniques are often too unpredictable (in their temporal behaviour) to be analysed. Attention in this section is therefore focused upon the interrupt-driven program-controlled and status-driven approaches.

Where an interrupt releases a sporadic task for execution, there is a cost that must be allocated to the interrupt handler itself. The priority of the handler is likely to be greater than that of the sporadic task, which means that tasks with priority greater than the sporadic task (but less than the interrupt handler) will suffer an interference. Indeed, this is an example of priority inversion, as the handler’s only job is to release the sporadic task – its priority should ideally be the same as the sporadic task. Unfortunately, most hardware platforms require the interrupt priorities to be higher than the ordinary software priorities. To model the interrupt handler, an extra ‘task’ is included in the schedulability test. It has a ‘period’ equal to the sporadic task, a priority equal to the interrupt priority level and an execution time equal to its own worst-case behaviour.

With status-driven devices, the control code can be analysed in the usual way. Such devices, however, do introduce a particular difficulty. Often the protocol for using an input device is as follows: ask for a reading, wait for the reading to be taken by the

hardware, and then access a register to bring the reading into the program. The problem is how to manage the delay while the reading is being taken. Depending on the likely duration of the delay, three approaches are possible:

- busy-wait on the 'done' flag;
- reschedule the task to some future time;
- for periodic tasks, split the action between periods.

With small delays, a busy-wait is acceptable. From a scheduling point of view, the 'delay' is all computation time and hence as long as the 'delay' is bounded, there is no change to the analysis approach. To protect against a failure in the device (that is, it never sets the done bit), a timeout algorithm can be used.

If the delay is sizeable, it is more effective to suspend the task, execute other work and then return to the I/O task at some future time when the value should be available. So if the reading time was 30 ms, the code would be:

```
begin
  --set up reading
  delay Milliseconds(30);
  -- take reading and use
end;
```

From a scheduling perspective, this structure has three significant implications. Firstly, the response times are not as easy to calculate. Each half of the task must be analysed separately. The total response time is obtained by adding together both sub-response times, and the 30 ms delay. Although there is a delay in the task, this is ignored when considering the impact that this task has on lower-priority tasks. Secondly, the extra computation time involved in delaying and being rescheduled again must be added to the worst-case execution time of the task (see Section 11.16 for a discussion on how to include system overheads). Thirdly, there is an impact on blocking. Recall that the simple equation for calculating the response time of a task is (see Section 11.8.1):

$$R_i = C_i + B_i + I_i$$

B_i is the blocking time (that is, the maximum time the task can be delayed by the actions of a lower-priority task). Various protocols for resource sharing were considered in Section 11.8. The effective ones all had the property that B_i consisted of just one block. However, when a task delays (and lets lower-priority tasks execute), it can be blocked again when it is released from the delay queue. Hence the response time equation becomes:

$$R_i = C_i + (N + 1)B_i + I_i$$

where N is the number of internal delays.

With periodic tasks, there is another way of managing this explicit delay. This method is called **period displacement** and involves initiating the reading in one period but taking the reading in the next. For example:

```
-- set up first reading loop
delay until Next_Release;
-- check done flag set
```



```

-- take reading and use
-- set up for next reading
Next_Release := Next_Release + Period;
end loop;

```

This is a straightforward approach with no impact on scheduling. Of course, the reading is one period old, which may not be acceptable to the application. To ensure that there is sufficient gap between the end of one execution and the start of the next, the deadline of the task can be adjusted. So, if S is the settling time for the device the required constraint is $D \leq T - S$. Note that the maximum staleness of the reading is bounded by $T + D$ (or $T + R$ once the worst-case response time is calculated).

14.7 Memory management

Embedded real-time systems often have only a limited amount of memory available; this is either because of the cost or because of other constraints associated with the surrounding system (for example, size, power or weight constraints). It may, therefore, be necessary to control how this memory is allocated so that it can be used effectively. Furthermore, where there is more than one type of memory (with different access characteristics) within the system, it may be necessary to instruct the compiler to place certain data types at certain locations. By doing this, the program is able to increase performance and predictability as well as interact with the outside world.

This chapter has already considered how data items can be allocated to particular memory locations, and how certain fields within storage units can be used to represent specific data types. This section considers the more general issues of storage management. Attention is focused on management of the two basic components that compilers use to manage data at run-time: the heap and the stack.

14.7.1 Heap management

The run-time implementations of most programming languages provide a large amount of memory (called the **heap**) so that the programmer can make run-time requests for chunks to be allocated (for example, to contain an array whose bounds were not known at compile time). An allocator (usually the **new** operator) is used for this purpose. It returns a pointer to memory within the heap of adequate size for the program's data structure. The run-time support system is responsible for managing the heap. Key problems are deciding how much space is required and when allocated space can be released. The first of these problems, in general, requires application knowledge. The second can be handled in several ways, including:

- require the programmer to return the memory explicitly – this is error-prone but is easy to implement; it is the approach taken by the C programming language where the functions `malloc` and `free` are used – the function `sizeof` allows the size of data types in bytes to be obtained (from the compiler);
- require the run-time support system to monitor the memory and determine when it can logically no longer be accessed – the scope rules of Ada allow an

implementation to adopt this approach; when an access type goes out of scope, all the memory associated with that access type can be freed;

- require the run-time support system to monitor the memory, and release chunks which are no longer being used (**garbage collection**) – this is, perhaps, the most general approach, as it allows memory to be freed even though its associated access type is still in scope.

From a real-time perspective, the above approaches have an increasing impact on the ability to analyse the timing properties of the program. In particular, garbage collection may be performed either when the heap is empty or by an asynchronous activity (incremental garbage collection). In either case, running the garbage collector may have a significant impact on the response time of a time-critical task. Although there has been much work on real-time garbage collection and progress continues to be made, there is still a reluctance to rely on these techniques in time-critical systems.

The following subsections will explore in more depth the facilities provided by Ada and Real-Time Java.

Heap management in Ada

In Ada, the heap is represented by one or more **storage pools**. Storage pools are associated with a particular Ada partition (for a non-distributable Ada system this is simply the whole program). Each object which is of an access type has an associated storage pool. The allocator ('new') takes its memory from the target pool. The `Ada.Unchecked_Deallocation` facility returns data to the pool. An implementation may support a single global pool which will be reclaimed when the partition terminates, or it may support pools defined at different accessibility levels which will be reclaimed when the associated scope is exited. By default, the implementation chooses a standard storage pool per access type. Note that all objects accessed directly (not via a pointer) are placed on the stack, not the heap.

To give more user control over storage management, Ada defines a package called `System.Storage_Pools` which is given in Program 14.5.

Programmers can implement their own storage pools by extending the `Root_Storage_Pool` type and providing concrete implementations for the subprogram bodies. To associate an access type with a storage pool, the pool is first declared and then the `Storage_Pool` attribute is used:

```
My_Pool : Some_Storage_Pool_Type;

type A is access Some_Object;
for A'Storage_Pool use My_Pool;
```

Now all calls to 'new' using A will automatically call `Allocate`; calls to `Ada.Unchecked_Deallocation` will call `Deallocate`; both referring to `My_Pool`. Furthermore, the implementation will call `Deallocate` when the access type goes out of scope.

Finally, it should be noted that Ada does not require an implementation to support garbage collection. However, it does support a pragma `Controlled` which indicates that garbage collection on a particular type should *not* be performed.

Program 14.5 The Ada System.Storage_Pools package.

```

with Ada.Finalization; with System.Storage_Elements;
package System.Storage_Pools is
  pragma Preelaborate(System.Storage_Pools);

  type Root_Storage_Pool is abstract new
    Ada.Finalization.Limited_Controlled with private;

  procedure Allocate(Pool : in out Root_Storage_Pool;
    Storage_Address : out Address;
    Size_In_Storage_Elements : in System.
      Storage_Elements.Storage_Count;
    Alignment : in System.Storage_Elements.Storage_Count)
    is abstract;

  procedure Deallocate(Pool : in out Root_Storage_Pool;
    Storage_Address : in Address;
    Size_In_Storage_Elements : in System.
      Storage_Elements.Storage_Count;
    Alignment : in System.Storage_Elements.Storage_Count)
    is abstract;

  function Storage_Size(Pool : Root_Storage_Pool) return
    System.Storage_Elements.Storage_Count is abstract;
private
  ...
end System.Storage_Pools;

```

Heap management in Real-Time Java

In contrast to Ada, all objects in Java are allocated on the heap and the language requires garbage collection for an effective implementation. Real-Time Java recognizes that it is necessary to allow memory management which is not affected by the vagaries of garbage collection. To this end, it introduces the notion of **memory areas**, some of which exist outside the traditional Java heap and never suffer garbage collection. Program 14.6 illustrates some of the main facilities of the abstract class for all memory areas. When a particular memory area is entered, all object allocation is performed within that area.

Using this abstract class, Real-Time Java defines various kinds of memory including the following.

- **Immortal memory** – immortal memory is shared among all threads in an application. Objects created in immortal memory are never subject to garbage collection and are freed only when the program terminates.

```

public final class ImmortalMemory extends MemoryArea {
  public static ImmortalMemory instance();
}

```

Program 14.6 An abridged MemoryArea class.

```

public abstract class MemoryArea {
    protected MemoryArea(long sizeInBytes);
    ...

    public void enter(java.lang.Runnable logic);
    // associate this memory area to the current thread
    // for the duration of the logic.run method

    public static MemoryArea getMemoryArea(java.lang.Object object);
    // get the memory area associated with the object

    public long memoryConsumed();
    // number of bytes consumed in this memory area
    public long memoryRemaining();
    // number of bytes remaining
    public long size(); // the size of the memory area
    ...
}

```

There is also a class called `ImmortalPhysicalMemory` which has the same characteristics as immortal memory but allows objects to be allocated from within a range of physical addresses.

- **Scoped memory** – scoped memory is a memory area where objects which have a well-defined lifetime can be allocated. A scoped memory may be entered explicitly (by the use of the `enter` method) or implicitly by attaching it to a `RealtimeThread` at thread creation time. Associated with each scoped memory is a reference count. This is incremented for every call to `enter` and at every associated thread creation. It is decremented when the `enter` method returns and at every associated thread exit. When the reference count reaches 0, all objects resident in the scoped memory have their finalization method executed and the memory is reclaimed. Scoped memory can be nested by nested calls to the `enter` method.

The `ScopedMemory` class (defined in Program 14.7) is an abstract class which has several subclasses including

- `VTMemory` – allocations may take variable amounts of time;
- `LTMemory` – allocations occur in linear time (related to the size of the object);
- `ScopedPhysicalMemory` – allowing objects to be allocated at physical memory locations.

To avoid the possibility of dangling pointers, a set of access restrictions are placed on the use of the various memory areas.

- **Heap objects** – can reference other heap objects and objects in immortal memory only (i.e. it cannot access scoped memory).

Program 14.7 The Real-Time Java ScopedMemory class.

```
public abstract class ScopedMemory extends MemoryArea {
    public ScopedMemory(long size);

    public void enter(java.lang.Runnable logic);

    public int getMaximumSize();

    public MemoryArea getOuterScope();

    public java.lang.Object getPortal();

    public void setPortal(java.lang.Object object);
}
```

- **Immortal objects** – can reference heap objects and immortal memory objects only.
- **Scoped objects** – can reference heaped objects, immortal objects and objects in the same scope or an outer scope only.

Memory parameters can be given when real-time threads and asynchronous event handlers are created. They can be used by the scheduler as part of an admission control policy and/or for the purpose of ensuring adequate garbage collection. Program 14.8 defines the class.

Consider, for example, a real-time thread which wishes to have all its memory allocated, by default, from immortal memory. However, during certain parts of its computation it wishes to create some temporary objects with a well-defined lifetime. First,

Program 14.8 The Real-Time Java MemoryParameters class.

```
public class MemoryParameters {
    public static final long NO_MAX;

    public MemoryParameters(long maxMemoryArea, long maxImmortal)
        throws IllegalArgumentException;

    public MemoryParameters(long maxMemoryArea, long maxImmortal,
        long allocationRate)
        throws IllegalArgumentException;

    public long getAllocationRate();
    public long getMaxImmortal();
    public long getMaxMemoryArea();

    public void setAllocationRate(long rate);
    public boolean setMaxImmortal(long maximum);
    public boolean setMaxMemoryArea(long maximum);
}
```

the code for the thread is defined. The method `computation` represents the part of the code where temporary storage is required. It creates some linear time-scoped memory indicating the minimum and maximum size it requires. It then defines a local `Runnable` to contain the actual computation. The code `myMem.enter` limits the scope of the memory.

```
import javax.realtime.*;
public class ThreadCode implements Runnable {
    private void computation() {
        final int min = 1*1024;
        final int max = 1*1024;
        final LTMemory myMem = new LTMemory(min, max);

        myMem.enter(new Runnable()
        {
            public void run() {
                // code here which requires access
                // to temporary memory
            }
        });
    }

    public void run() {
        ...
        computation();
        ...
    }
}
```

The thread can now be created. Note that in this example no parameters other than a `MemoryArea` object and a `Runnable` object are given.

```
ThreadCode code = new ThreadCode();

RealtimeThread myThread = new RealtimeThread(
    null, null, null, ImmortalMemory.instance(),
    null, code);
```

14.7.2 Stack management

As well as managing the heap, embedded programmers also have to be concerned with stack size. While specifying the stack size of a task/thread requires trivial support (for example, in Ada it is via the `Storage_Size` attribute applied to a task; in POSIX it is via `pthread` attributes), calculating the stack size is more difficult. As tasks enter blocks and execute procedures their stacks grow. To accurately estimate the maximum extent of this growth requires knowledge of the execution behaviour of each task. This knowledge is similar to that required to undertake worst-case execution time (WCET) analysis (see Section 11.13). Hence both WCET and worst-case stack usage bounds can be obtained from a single tool performing control flow analysis of the task's code.

Summary

One of the main characteristics of an embedded system is the requirement that it interacts with special-purpose input and output devices. To program device drivers in high-level languages requires:

- the ability to pass data and control information to and from the device;
- the ability to handle interrupts.

Normally control and data information is passed to devices through device registers. These registers are either accessed by special addresses in a memory-mapped I/O architecture, or via special machine instructions. Interrupt handling requires context switching, device and interrupt identification, interrupt control and device prioritization.

The programming of devices has traditionally been the stronghold of the assembly language programmer, but languages like C, Java and Ada have progressively attempted to provide high-level mechanisms for these low-level functions. This makes device driver and interrupt-handling routines easier to read, write and maintain. The main requirement on a high-level language is that it provides an abstract model of device handling. Encapsulation facilities are also required so that the non-portable code of the program can be separated from the portable part.

The model of device handling is built on top of the language's model of concurrency. A device can be considered to be a processor performing a fixed task. A computer system can therefore be modelled as several parallel tasks which need to communicate and synchronize. There are several ways in which interrupts can be modelled. They must all have:

- (1) facilities for addressing and manipulating device registers;
- (2) a suitable representation of interrupts.

In Ada, device registers can be defined as scalars and user-defined record types, with a comprehensive set of facilities for mapping types onto the underlying hardware. Interrupts are viewed as hardware-generated procedure calls to a protected object.

Real-Time Java supports the access to memory-mapped I/O registers through the `RawMemoryAccess` class; however, it lacks expressive power for manipulating device registers. Interrupts are associated with asynchronous events.

Low-level programming also involves the more general issue of managing the memory resources of the processor. This chapter has considered both stack and heap management. Ada does not require a garbage collector – memory can be explicitly deallocated and the scope rules of the language allow automatic deallocation when an access type goes out of scope. The language also allows user-defined storage pools to be defined which enable programmers to define their own memory management policies.

Real-Time Java recognizes that the memory allocation policy of Java is not sustainable for real-time systems. Consequently, it allows memory to be allocated outside of the heap, and supports the notion of scoped memory which allows automatic reclamation of memory without garbage collection.

Further reading

- Barr, M. (1999) *Programming Embedded Systems in C and C++*. Sebastopol, CA: O'Reilly.
- Burns, A. and Wellings, A. J. (2007) *Concurrent and Real-time Programming in Ada*. Cambridge: Cambridge University Press.
- Dibble, P. C. (2008) *Real-Time Java Platform Programming*, 2nd edn, BookSurge Publishing, www.booksurge.com.
- Project UDI (2008) *Uniform Driver Interface (UDI)* <http://www.projectudi.org/>, accessed September 2008.
- Regehr, R. (2007) Safe and structured use of interrupts in real-time and embedded software, in *Handbook of Real-Time and Embedded Systems*, I. Lee, J. Y-T. Leung and S. H. Son (eds). Boca Raton, FL: Chapman and Hall/CRC.
- Wellings, A. J. (2004) *Concurrent and Real-Time Programming in Java*. Chichester: Wiley.

Exercises

- 14.1** Consider a computer which is embedded in a patient-monitoring system (assume the simple I/O system given in this chapter). The system is arranged so that an interrupt is generated at the highest hardware priority, through vector location 100 (octal) every time the patient's heart beats. In addition, a mild electric shock can be administered via a device control register, the address of which is 177760 (octal). The register is set up so that every time an integer value x is assigned to it the patient receives x volts over a small period of time.

If no heartbeat is recorded within a 5 second period the patient's life is in danger. Two actions should be taken when the patient's heart fails: the first is that a 'supervisor' task should be notified so that it may sound the hospital alarm, and the second is that a single electric shock of 5 volts should be administered. If the patient fails to respond, the voltage should be increased by 1 volt for every further 5 seconds.

Write an Ada program which monitors the patient's heart and initiates the actions described above. You may assume that the supervisor task is given by the following specification:

```
task Supervisor is
  entry Sound_Alarm;
end Supervisor;
```

- 14.2** The Spanish government is considering introducing charges for the use of motorways. One possible mechanism is that detector stations can be set up at regular

intervals along all motorways; as a vehicle passes the detector station its details are logged and a road charge is recorded. At the end of each month the vehicle owner can be sent a bill for his or her motorway usage.

Each vehicle will require an interface device which interrupts an on-board computer when a detector station requests its details. The computer has a word size of 16 bits and has memory-mapped I/O, with all I/O registers 16 bits in length. Interrupts are vectored, and the address of the interrupt vector associated with the detector station is 8#60#. After an interrupt has been received, a read-only input register located at 8#177760# contains the basic cost of using the current stretch of motorway (in euros). The hardware priority of the interrupt is 4.

The computer software must respond to the interrupt within a period of 5 seconds, and pass its ownership details to the detector station via the interface device. It does this by writing to a bank of five control and status registers located at 8#177762#. The structure of these registers is shown in Table 14.1.

Register	Bits	Meaning
1	0–7	Vehicle registration character 1
1	8–15	Vehicle registration character 2
2	0–7	Vehicle registration character 3
2	8–15	Vehicle registration character 4
3	0–7	Vehicle registration character 5
3	8–15	Vehicle registration character 6
4	0–7	Vehicle registration character 5
4	8–15	Vehicle registration character 8
5	0	set to 1 to transmit data
5	1–4	travel details
		1 = business
		2 = pleasure
		3 = overseas tourist
		4 = police
		5 = military
		6 = emergency services
5	5–15	security code (0–2047)

Table 14.1 Register structure for road charging.

The bank of ‘control and status’ registers (CSR) is write-only.

Write an Ada task to interface with the detector station. The task should respond to interrupts from the interface device and be responsible for sending the correct vehicle details. It should also read the data register containing the cost of the road usage, and pass the current total cost of the journey to a task which will output it on a visual display unit on the vehicle’s dashboard. You may assume the following is available:

```
package Journey_Details is
  Registration_Number : constant String(1..8) :=
    ".....";
```

```

type Travel_Details is
    (Business, Pleasure, Overseas_Tourist,
     Police, Military, Emergency_Service);
for Travel_Details use
    (Business => 1, Pleasure => 2,
     Overseas_Tourist => 3, Police => 4,
     Military => 5 , Emergency_Service => 6);

subtype Security_Code is Integer range 0 .. 2047;

function Current_Journey return Travel_Details;

function Code return Security_Code;
end Journey_Details;

package Display_Interface is

    task Display_Driver is
        entry Put_Cost(C : Integer);
        -- prints the cost on the dashboard display
    end Display_Driver;

end Display_Interface;

```

Assume that the compiler represents the type `Character` as an 8-bit value and the type `Integer` as a 16-bit value.

14.3 Rewrite your answer to Exercise 14.2 in Real-Time Java.

14.4 The British government is concerned about the speed of cars using motorways. In the future, beacons will be set up at regular intervals along all motorways; they will continuously transmit the current speed limit. New cars will contain computers which will monitor the current speed limit and inform the driver when he or she exceeds the limit.

One car currently being designed (the Yorkmobile) already has the necessary hardware interfaces. They are as follows.

- Each car has a 'speed control' 16-bit computer which has memory-mapped I/O, with all I/O registers 16 bits in length.
- A register located at octal location 177760 interfaces to a device which monitors the roadside beacons. The register always contains the value of the last speed limit received from the roadside beacons.
- A pair of registers interface to an intelligent speedometer device which checks the speed of the car against a set limit. If the speed limit is broken, the device generates an interrupt through octal location 60. The priority of the interrupt is 5. This interrupt is repeated every 5 seconds until the car is no longer speeding.
- The register pair consists of a CSR, and a data buffer register (DBR). The structure of the CSR register is shown in Table 14.2.

The CSR register may be both read from and written to, and resides at address octal 177762. The DBR register simply contains an integer value representing the car's speed limit to be set. If this value falls outside the

Bits	Meaning
0	enable device
1	when set the value found in the DBR is used as the car's current speed limit
5-2	not used
6	interrupt enable
11-7	not used
15-12	error bits (0 = no error, > 0 illegal limit)

Table 14.2 Control register structure speed computer.

range 0 – 70 then an illegal limit has been specified, and the device continues with the current limit. The address of the DBR register is octal 177764.

- A flashing light (on the car dashboard) can be turned on by setting the register, located at octal address 177750, to 1. The light will flash for 5 seconds only. A zero value turns the light off.

Design an Ada and Real-Time Java device driver which will implement the following speed control algorithm.

Every 60 seconds the current speed limit should be read from the roadside beacon by the speed control computer. This value is passed unchecked to the speedometer device which interrupts if the car exceeds the set speed limit or if the speed limit is illegal. If the car exceeds the limit then the dashboard light should be flashed until the car returns to the current limit.

Chapter 15

Mine control case study

15.1	Mine drainage	15.5	Translation to Ada
15.2	The HRT-HOOD design method	15.6	Translation to Real-Time Java
15.3	The logical architecture design	15.7	Fault tolerance and distribution
15.4	The physical architecture design		Summary
			Further reading
			Exercises

In this chapter, a case study is presented which includes many of the facilities described in this book. A full design is given using the HRT-HOOD design method and the UML 2.0 diagrammatical notation. Implementations in Ada and Real-Time Java are then derived from systematic translations. As the goal of this chapter is to illustrate the real-time programming abstractions, these translations are kept as simple as possible.

A full implementation is shown for the real-time components in Ada, whilst because of space constraints only elements of the system will be given in Real-Time Java. The translation to C/Real-Time POSIX is left as an exercise for the reader.

15.1 Mine drainage

The example that has been chosen is based on one which commonly appears in the literature. It concerns the software necessary to manage a simplified pump control system for a mining environment (Kramer et al., 1983; Shrivastava et al., 1987; Sloman and Kramer, 1987; Burns and Lister, 1991; Joseph, 1996; de la Puente et al., 1996); it possesses many of the characteristics which typify embedded real-time systems. It is assumed that the system will be implemented on a single processor with a simple memory-mapped I/O architecture.

The system is used to pump water, which collects in a sump at the bottom of the mine shaft, to the surface. The main safety requirement is that the pump should not be operated when the level of methane gas in the mine reaches a high value due to the risk of explosion. A simple schematic diagram of the system is given in Figure 15.1.

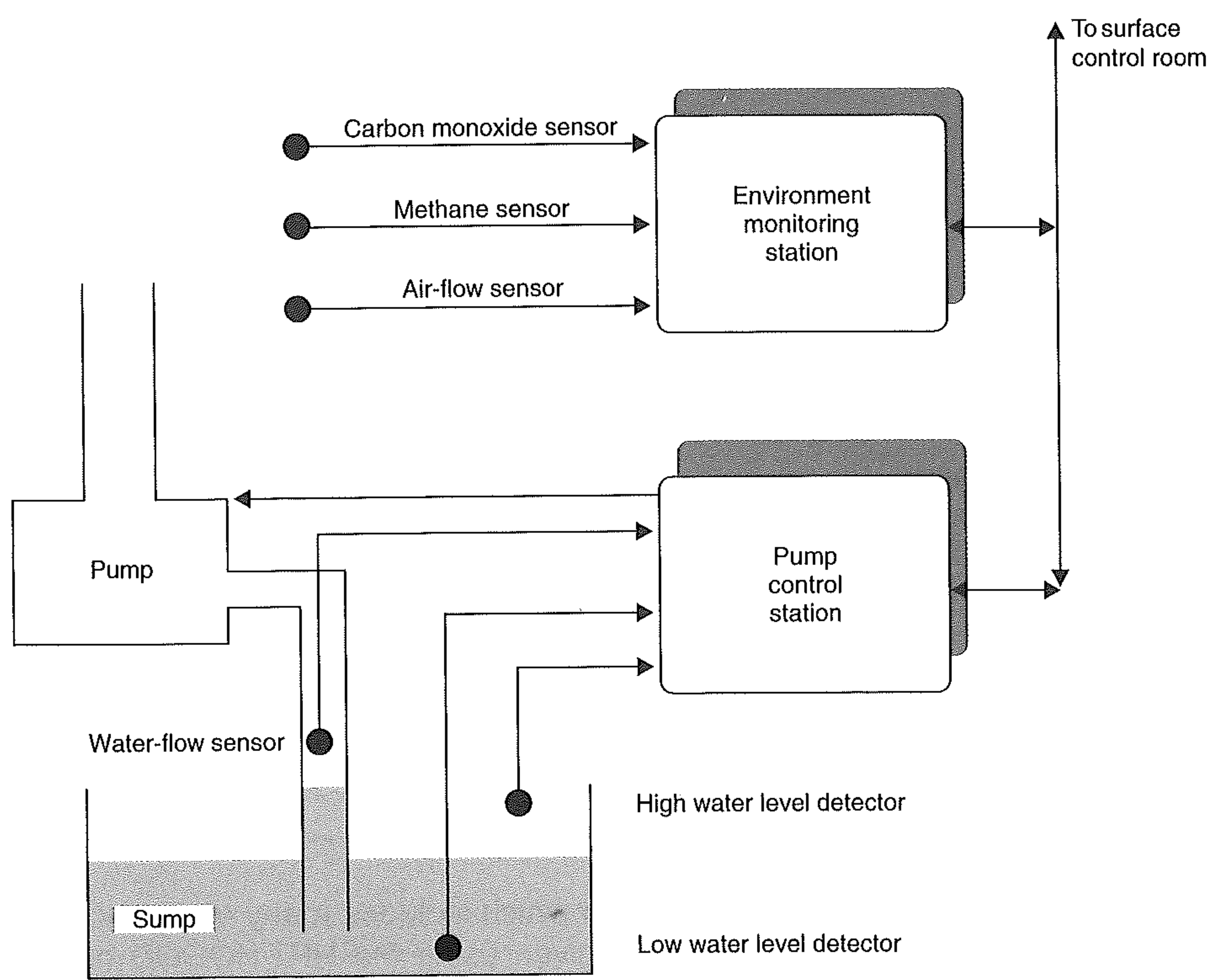


Figure 15.1 A mine drainage control system.

The relationship between the control system and the external devices is shown in Figure 15.2. Note that only the high and low water sensors communicate via interrupts (indicated by dashed arrows); all the other devices are either polled or directly controlled.

Most traditional software development methods incorporate a life cycle model in which the following activities are recognized:

- **requirements specification** – during which an authoritative specification of the system’s required functional and meta-functional behaviour is produced;
- **architectural design** – during which a top-level description of the proposed system is developed;
- **detailed design** – during which the complete system design is specified;
- **coding** – during which the system is implemented;
- **testing** – during which the efficacy of the system is tested.

For hard real-time systems, this has the significant disadvantage that timing problems will only be recognized during testing, or worse, after deployment. HRT-HOOD (Burns and Wellings, 1995) is different from traditional design methods in that it directly addresses the concerns of hard real-time systems. It views the design process

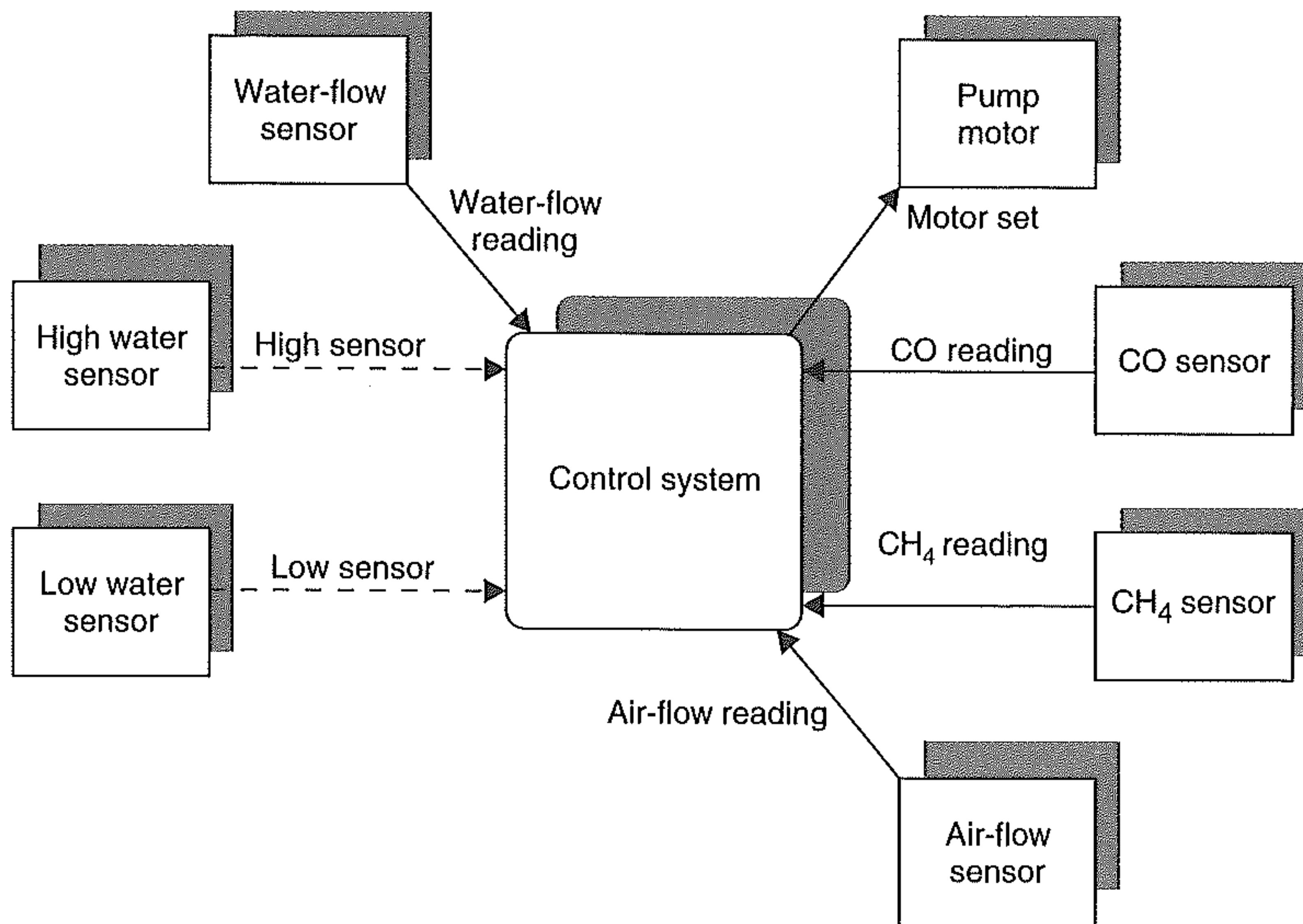


Figure 15.2 Graph showing external devices.

as a progression of increasingly specific *commitments* (Burns and Lister, 1991). These commitments define properties of the system design which designers operating at a more detailed level are not at liberty to change. Those aspects of a design to which no commitment is made at some particular level in the design hierarchy are effectively the subject of *obligations* that lower levels of design must address. Early in design there may already be commitments to the architectural structure of a system, in terms of object definitions and relationships. However, the detailed behaviour of the defined objects remains the subject of obligations which must be met during further design and implementation.

The process of refining a design – transforming obligations into commitments – is often subject to *constraints* imposed primarily by the execution environment. The execution environment is the set of hardware and software components (for example, processors, task dispatchers, device drivers) on top of which the system is built. It may impose both resource constraints (for example, processor speed, communication bandwidth) and constraints of mechanism (for example, interrupt priorities, task dispatching, data locking). To the extent that the execution environment is immutable, these constraints are fixed.

Obligations, commitments and constraints have an important influence on the architectural design of any application. Therefore HRT-HOOD defines two activities of the architectural design:

- the logical architecture design activity;
- the physical architecture design activity.

The logical architecture embodies commitments which can be made independently of the constraints imposed by the execution environment, and is primarily aimed at satisfying

the functional requirements (although the existence of timing requirements, such as end-to-end deadlines, will strongly influence the decomposition of the logical architecture). The physical architecture takes these functional requirements and other constraints into account, and embraces the meta-functional requirements. The physical architecture forms the basis for asserting that the application's meta-functional requirements will be met once the detailed design and implementation have taken place. It addresses timing and dependability requirements, and the necessary schedulability analysis that will ensure (guarantee) that the system, once built, will function correctly in both the value and time domains.

Although the physical architecture is a refinement of the logical architecture, its development will usually be an iterative and concurrent process in which both models are developed/modified. The analysis techniques embodied in the physical architecture can, and should, be applied as early as possible. Initial resource budgets can be defined that are then subject to modification and revision as the logical architecture is refined. In this way a 'feasible' design is tracked from requirements through to deployment.

In this chapter, the HRT-HOOD method is followed, but UML 2.0 is used to represent the logical and physical architectures.

15.1.1 Functional requirements

The functional specification of the system may be divided into four components: the pump operation, the environment monitoring, the operator interaction and system monitoring.

Pump operation

The required behaviour of the pump controller is that it monitors the water levels in the sump. When the water reaches a high level (or when requested by the operator), the pump is turned on and the sump is drained until the water reaches the low level. At this point (or when requested by the operator), the pump is turned off. A flow of water in the pipes can be detected if required.

The pump should only be allowed to operate if the methane level in the mine is below a critical level. An alarm should be signalled if the pump cannot operate or if there is no water flow when there should be.

Environment monitoring

The environment must be monitored to detect the level of methane in the air; there is a level beyond which it is not safe to cut coal or operate the pump. The monitoring also measures the level of carbon monoxide in the mine and detects whether there is an adequate flow of air. Alarms must be signalled if gas levels or air flow become critical.

Operator interaction

The system is controlled from the surface via an operator's console. The operator is informed of all critical events.

System monitoring

All the system events are to be stored in an archival database, and may be retrieved and displayed upon request.

15.1.2 Meta-functional requirements

The meta-functional requirements can be divided into three components: timing, dependability and security. This case study is mainly concerned with the timing requirements and consequently dependability and security will not be addressed (see Burns and Lister (1991) for a full consideration of dependability and security aspects).

There are several requirements which relate to the timeliness of system actions. The following list is adapted from Burns and Lister (1991).

(i) Monitoring periods

The maximum periods for reading the environment sensors may be dictated by legislation. For the purpose of this example, it is assumed that these periods are the same for all sensors, namely 100 ms. In the case of methane, there may be a more stringent requirement based on the proximity of the pump and the need to ensure that it never operates when the methane level is critically high. This is discussed in (ii) below. In Section 14.6, it was described how a device driver can be analysed. In the case study, the 'period displacement' approach will be used for the CH₄ and CO sensors. These environmental sensors each require 40 ms in order for a reading to become available. Hence they require a deadline of 60 ms.

The water flow object executes periodically and has two roles. While the pump is operational it checks that there is a water flow; but while the pump is off (or disabled) it also checks that the water has stopped flowing. This latter check is used as confirmation that the pump has indeed been stopped. Due to a time lag in the flow of water, this object is given a period of 1 second, and it uses the results of two consecutive readings to determine the actual state of the pump. To make sure that two consecutive readings are actually one second apart (approximately), the object is given a tight deadline of 40 ms (that is, two readings will be at least 960 ms, but no more than 1040 ms, apart).

It is assumed that the water level detectors are event-driven and that the system should respond within 200 ms. The physics of the application indicates that there must be at least 6 seconds between interrupts from the two water level indicators.

(ii) Shut-down deadline

To avoid explosions, there is a deadline within which the pump must be switched off once the methane level exceeds a critical threshold. This deadline is related to the methane-sampling period, to the rate at which methane can accumulate, and to the margin of safety between the level of methane regarded as critical and the level at which it explodes. With a direct reading of the sensor, the relationship can be expressed by the inequality:

$$R(T + D) < M$$

where

R is the rate at which methane can accumulate

T is the sampling period

D is the shut-down deadline

M is the safety margin.

	Periodic/sporadic	'Period'	Deadline
CH ₄ sensor	P	80	30
CO sensor	P	100	60
Air flow	P	100	100
Water flow	P	1000	40
High water level detector	S	6000	200
Low water level detector	S	6000	200

Table 15.1 Attributes of periodic and sporadic entities.

If ‘period displacement’ is used then a further period of time is needed:

$R(2T + D) < M$

Note that the period T and the deadline D can be traded off against each other, and both can be traded off against the safety margin M. The longer the period or the deadline, the more conservative must be the safety margin; the shorter the period or deadline, the closer to its safety limits the mine can operate. The designer may therefore vary any of D, T or M in satisfying the deadline and periodicity requirements.

In this example, it is assumed that the presence of methane pockets may cause levels to rise rapidly, and therefore a deadline requirement (from methane going high to the pump been disabled) of 200 ms is assumed. This can be met by reducing the rate for the methane sensor to 80 ms, with a deadline of 30 ms. Note that this level will ensure that correct readings are taken from the sensor (that is, the displacement between two readings is at least 50 ms).

(iii) Operator information deadline

The operator must be informed within 1 second of detection of critically high methane or carbon monoxide readings, within 2 seconds of a critically low air-flow reading, and within 3 seconds of a failure in the operation of the pump. These requirements are easily met when compared with the other timing requirements.

In summary, Table 15.1 defines the periods, or minimum inter-arrival times (‘period’), and deadlines (in milliseconds) for the sensors.

15.2 The HRT-HOOD design method

HRT-HOOD facilitates the logical architectural design of a system by providing different object types. They are:

- **Passive** – re-entrant objects which have no control over when invocations of their operations are executed, and do not spontaneously invoke operations in other objects.

- **Active** – objects which may control when invocations of their operations are executed, and may spontaneously invoke operations in other objects. Active objects are the most general class of objects and have no restrictions placed on them.
- **Protected** – objects which may control when invocations of their operations are executed, but do not spontaneously invoke operations in other objects; in general protected objects may **not** have arbitrary synchronization constraints and must be analysable for the blocking times they impose on their callers.
- **Cyclic** – objects which represent periodic activities; they may spontaneously invoke operations in other objects and only have very restrictive interfaces.
- **Sporadic** – objects which represent sporadic activities; sporadic objects may spontaneously invoke operations in other objects; each sporadic has a **single** operation which is called to invoke the sporadic.

A hard real-time program designed using HRT-HOOD will contain at the terminal level (that is, after full design decomposition) only cyclic, sporadic, protected and passive objects. Active objects, because they cannot be fully analysed, are only allowed for background activity. Active object types may be used during decomposition of the main system but must be transformed into one of the other types before reaching the terminal level. HRT-HOOD supports the hierarchical decomposition of a *parent* object into *child* objects.

Within each cyclic and sporadic object, there will be a single task that is the entity subject to schedulability analysis as part of the verification of the physical architecture.

15.3 The logical architecture design

The logical architecture addresses those requirements which are independent of the physical constraints (for example, processor speed) imposed by the execution environment. The functional requirements identified in Section 15.1.1 fall into this category. Consideration of the other system requirements is deferred until the design of the physical architecture, described later.

15.3.1 First level decomposition

The first step in developing the logical architecture is the identification of appropriate subsystems from which the system can be built. The functional requirements of the system suggest four distinct components:

- (1) **pump controller** – responsible for operating the pump;
- (2) **environment monitor** – responsible for monitoring the environment;
- (3) **operator console** – responsible for the interface to the operators;
- (4) **data logger** – responsible for logging operational and environmental data.

Figure 15.3 illustrates this decomposition. Each component has a number of provided and required interfaces.

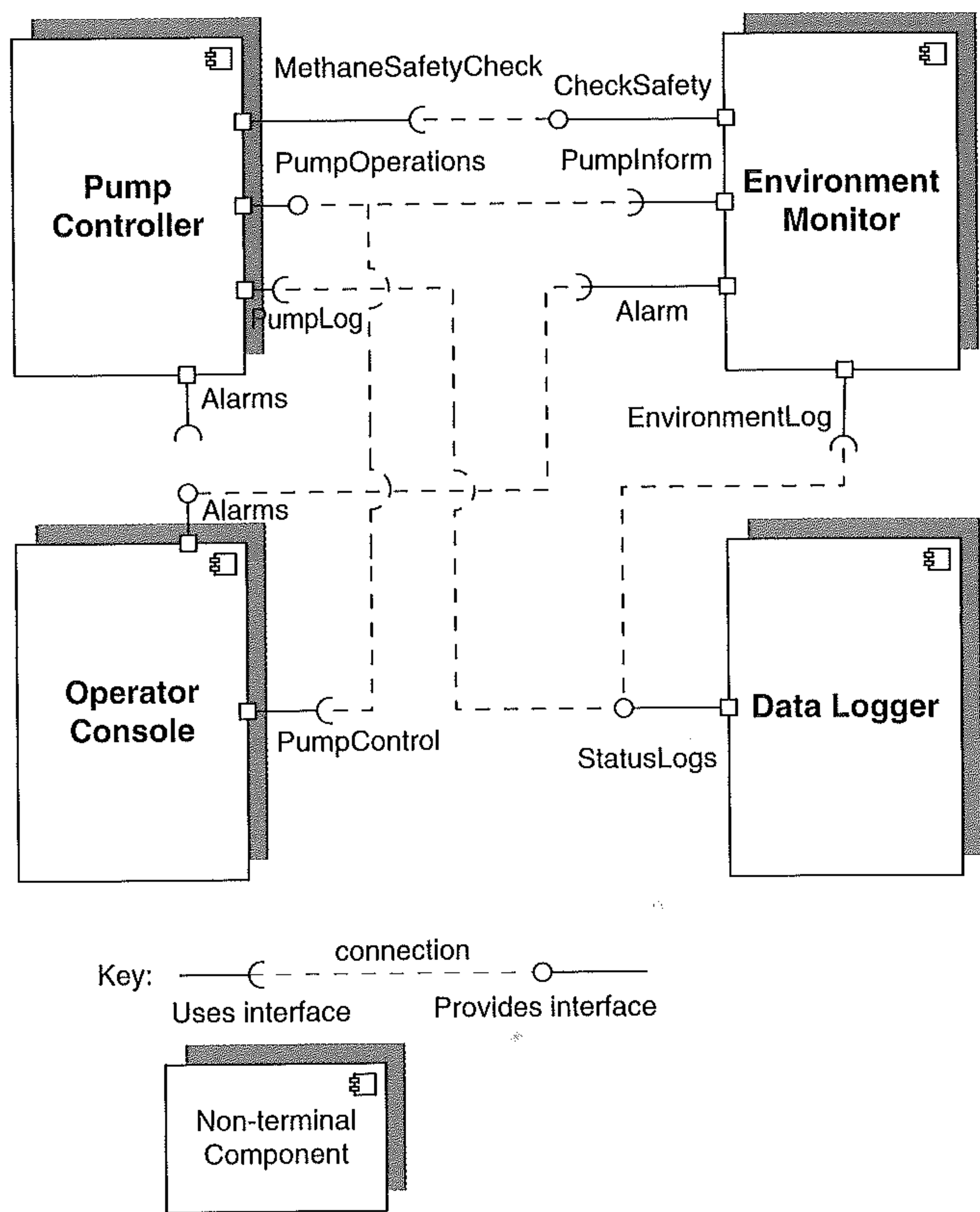


Figure 15.3 First-level component decomposition of the control system.

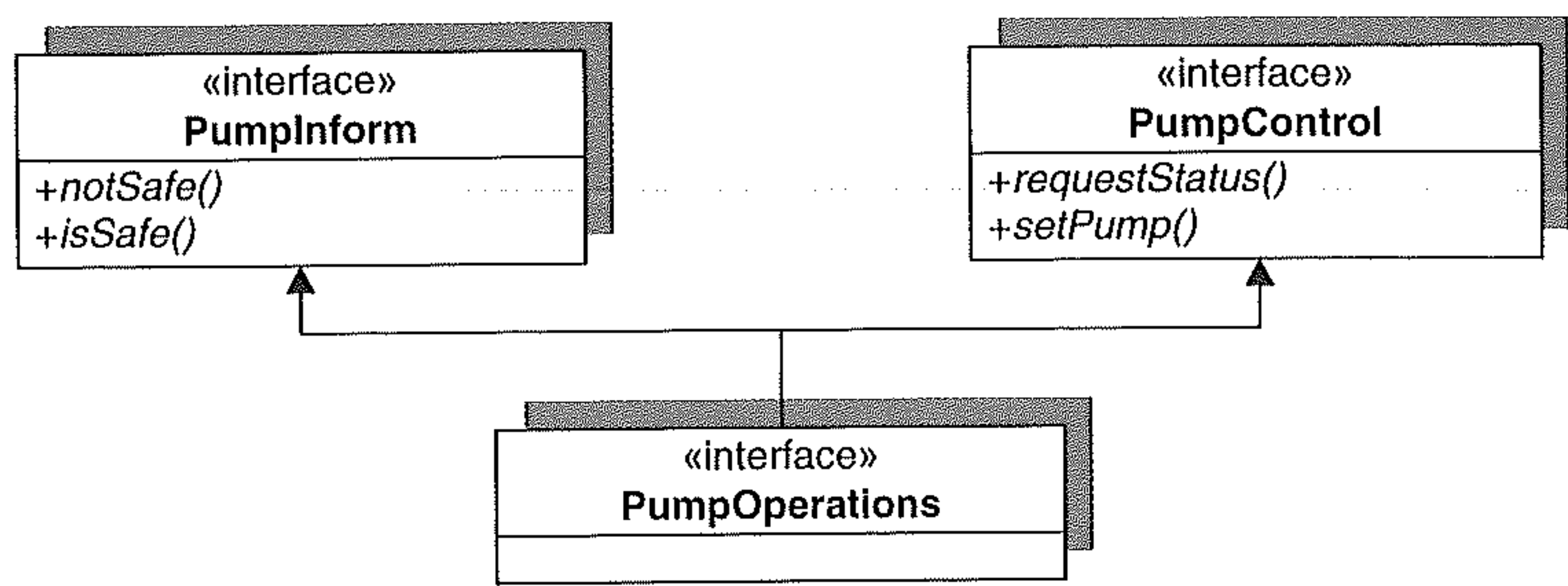


Figure 15.4 PumpController related interfaces.

- The PumpController provides one interface to the system (PumpOperations) which exports all the operations that are available on the pump. Both the OperatorConsole and the EnvironmentMonitor make use of these operations through their respective required interfaces. Figure 15.4 shows the relationship between these interfaces. The operations ‘not safe’ and ‘is safe’ are called by the EnvironmentMonitor to inform the PumpController

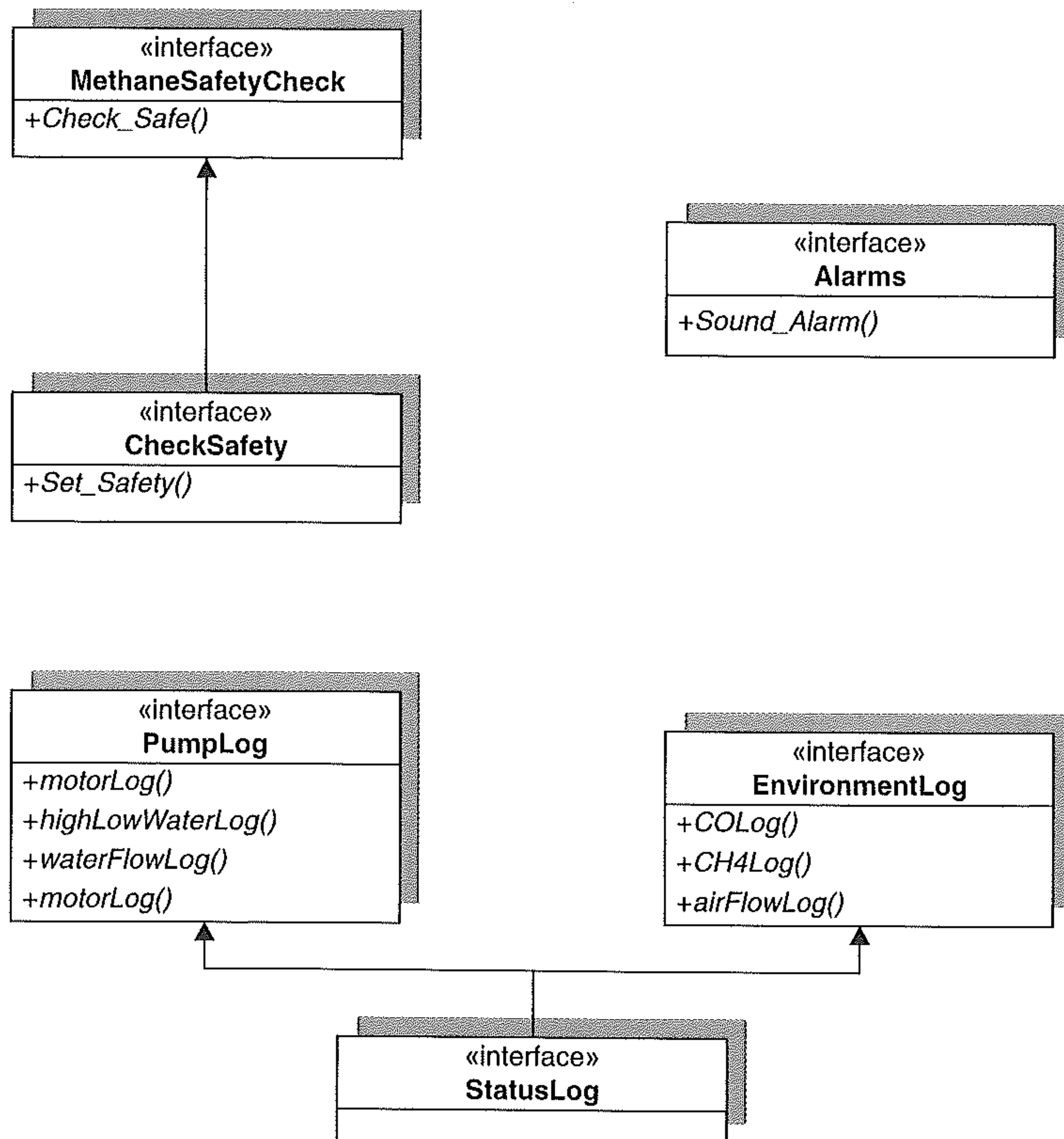


Figure 15.5 Other defined interfaces.

whether it is safe to operate the pump (due to the level of methane in the environment). The 'request status' and 'set pump' operations are called by the `OperatorConsole`.

- The `PumpController` makes use of the services provided by the other components via its `MethaneSafetyCheck`, `PumpLog` and `Alarms` required interfaces (see Figure 15.5 where all the other relationship between the various interfaces are shown). As an additional reliability feature, the pump controller will always check that the methane level is low before starting the pump (by calling 'check safe' operation via the `MethaneSafetyCheck` interface). If the pump controller finds that the pump cannot be started (or that the water does not appear to be flowing when the pump is notionally on) then it raises an alarm through the `Alarms` interface. The pump reports its state changes via the `PumpLog` interface.
- The `EnvironmentMonitor` has the single operation 'check safe' which it provides via the `CheckSafety` interface.
- The `OperatorConsole` provides the alarm operation via the `Alarms` interface, which as well as being called by the `PumpController` is also called by the `EnvironmentalMonitor` if any of its readings are too high. As well as receiving the alarm calls, the `OperatorConsole` can request the status of the pump and attempt to override the high and low water sensors by directly operating

the pump. However, in the latter case the methane check is still made, with an exception being used to inform the operator that the pump cannot be turned on.

- The DataLogger supports six operations which are merely data logging actions that are called by the pump controller and the environment monitor.

15.3.2 Pump controller

The decomposition appropriate to the pump controller is shown in Figure 15.6. The pump controller is decomposed into three objects. The first object controls the pump

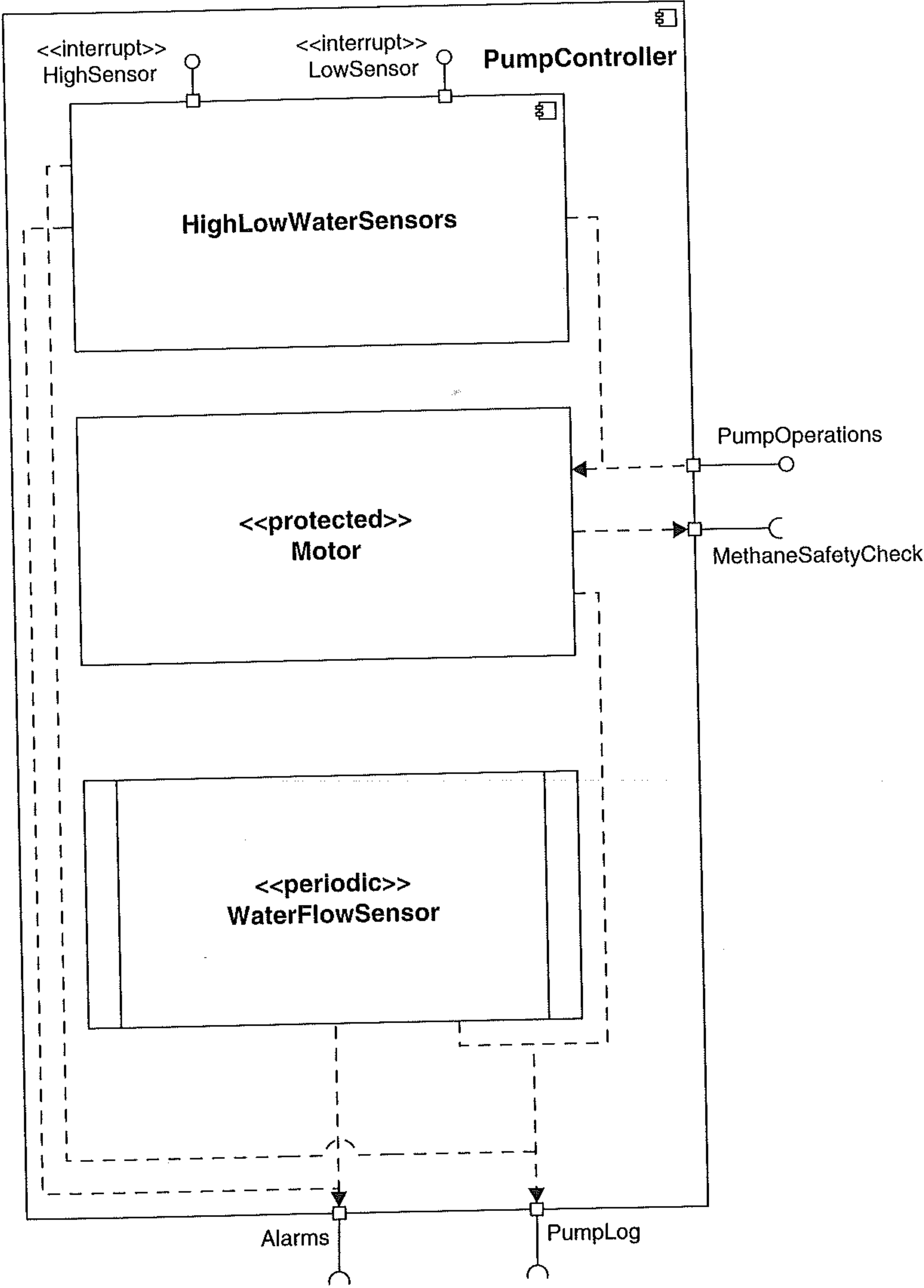


Figure 15.6 Hierarchical decomposition of the PumpController object.

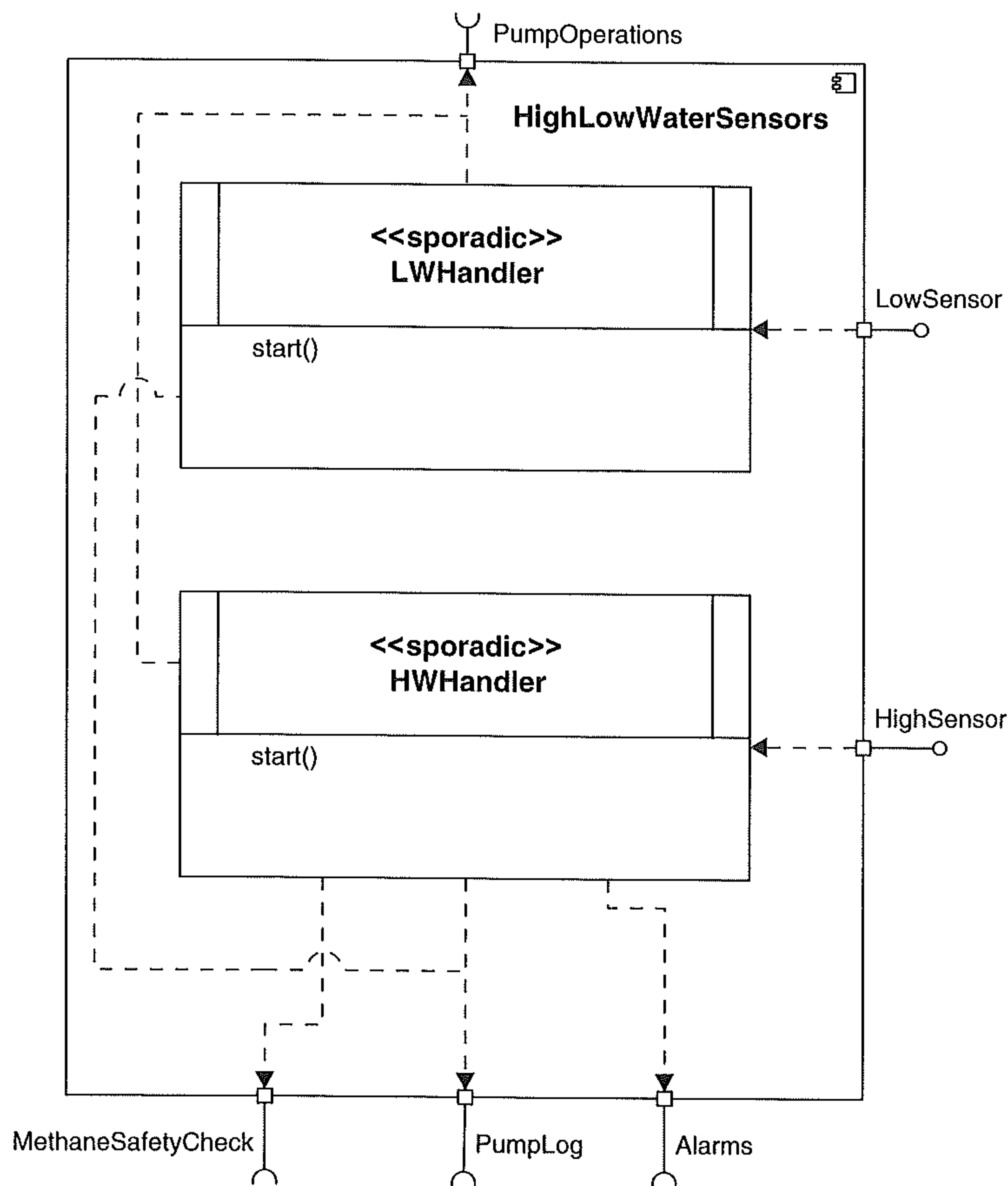


Figure 15.7 Decomposition of the HighLowWaterSensors.

motor. As this object simply responds to commands, requires mutual exclusion for its operations, and does not spontaneously call other objects, it is a *protected* object. All of the pump controller’s operations are implemented by the motor object. As the system is real-time, none of the operations can be arbitrarily blocked (although they require mutual exclusion).

The other two objects control the water sensors. The flow sensor object is a *cyclic* object which continually monitors the flow of water from the mine. The high–low water sensor is an *active* object which handles the interrupts from the high and low water sensors. It decomposes into two *sporadic* objects, as shown in Figure 15.7.

15.3.3 The environment monitor

The environment monitor decomposes into four terminal objects, as shown in Figure 15.8. Three of the objects are *cyclic* objects which monitor the CH₄ level, CO level and the air-flow in the mine environment. Only the CH₄ level is requested by other objects

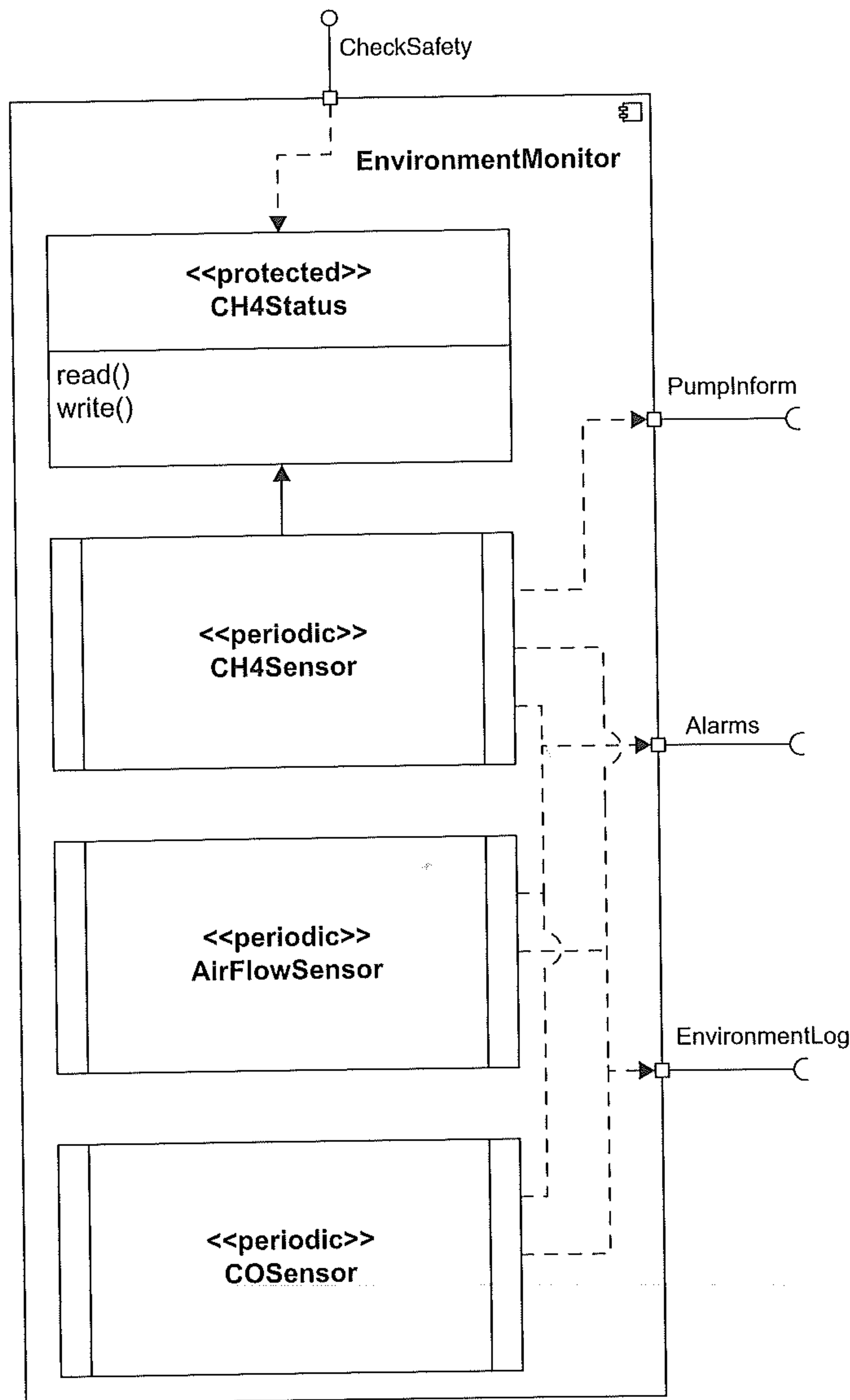


Figure 15.8 Hierarchical decomposition of the EnvironmentMonitor.

in the system; consequently a *protected* object is used to control access to the current value.

15.3.4 The data logger and the operator console

This case study is not concerned with the details of the data logger or the operator console. However, it is a requirement that they only delay the real-time threads for a bounded time. It is assumed, therefore, that their interfaces contain protected objects.

15.4 The physical architecture design

HRT-HOOD supports the design of a physical architecture by:

- allowing timing attributes to be associated with objects;
- providing a framework from within which a schedulability approach can be defined and the analysis of the terminal objects undertaken;
- providing the abstractions with which the designer can express the handling of timing errors.

To illustrate the analysis described in Chapter 11, fixed-priority scheduling will be used and the response time form of analysis will be undertaken. Table 15.2 summarizes the timing attributes of the objects introduced in the logical architecture.

15.4.1 Scheduling analysis

Once the code has been developed, it must be analysed to obtain its worst-case execution times. As indicated in Section 11.13, these values can be obtained either via direct measurement or by modelling the hardware. None of the code derived is particularly extensive, so it is reasonable to assume that a low-speed processor is adequate. Table 15.3 contains some representative values for the worst-case execution times (in milliseconds) for each object in the design. Note that the times for each task include time spent executing within other objects, the time spent executing exception handlers and the associated context switch times. To model the effect of the interrupt handlers 'pseudo' sporadic objects are introduced (the maximum handler execution time is 2 ms).

The execution environment imposes its own set of important parameters – these are given in Table 15.4. Note that the clock interrupt is of sufficient granularity to ensure no release jitter for the periodic tasks.

The maximum blocking time, for all tasks, occurs when the operator console makes a call upon the motor object. It can be assumed that the task that makes the call is of

	Type	'Period'	Deadline	Priority
CH ₄ sensor	Periodic	80	30	10
CO sensor	Periodic	100	60	8
Air-flow sensor	Periodic	100	100	7
Water-flow sensor	Periodic	1000	40	9
High water handler	Sporadic	6000	200	6
Low water handler	Sporadic	6000	200	6
Motor	Protected			10
CH ₄ status	Protected			10
Operator console	Protected			10
Data logger	Protected			10

Table 15.2 Attributes of design objects.

	Type	WCET
CH ₄ sensor	Periodic	12
CO sensor	Periodic	10
Air-flow sensor	Periodic	10
Water-flow sensor	Periodic	10
High water handler	Sporadic	20
Low water handler	Sporadic	20
Interrupt Low water	Sporadic	2
Interrupt High water	Sporadic	2

Table 15.3 Worst-case execution times.

	Symbol	Time
Clock period	T_{CLK}	20
Clock overhead	CT^c	2
Cost of single task move	CT^s	1

Table 15.4 Overheads.

	Type	T	B	C	D	P	R
CH ₄ sensor	Periodic	80	3	12	30	10	25
CO sensor	Periodic	100	3	10	60	8	47
Air-flow sensor	Periodic	100	3	10	100	7	57
Water-flow sensor	Periodic	1000	3	10	40	9	35
High water handler	Sporadic	6000	3	20	200	6	79
Low water handler	Sporadic	6000	3	20	200	6	79

Table 15.5 Analysis results.

low priority. The worst-case execution time for this protected operation is assumed to be 3 ms.

The above information can now be synthesized to provide a comprehensive analysis of the response times of all tasks in the system. This analysis is given in Table 15.5. The conclusion of the analysis is that all deadlines are met. Note that the two sporadic tasks are computed to have the same response times, as the interrupts that release the tasks are separated by at least 6 seconds. Hence, the sporadic tasks can never execute at the same time.

15.5 Translation to Ada

HRT-HOOD supports a systematic translation to Ada. For each non-terminal object, two packages are generated. The first contains all the types that are used for communication between its child objects. The second contains all the interfaces provided by that object. For each terminal object, two child packages are also generated: the first

simply contains a collection of data types and variables defining the object's real-time attributes; the second contains the code for the object itself (including a single task for cyclic and sporadic).

To ease the translation, two standard templates are used: one for a cyclic (periodic) task and one for a sporadic task. The periodic template is defined as a task type that takes an access discriminant to an extensible type. The application extends this type to provide the state and code for the task.

```
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
package Cyclics is
  type Cyclic_State is abstract tagged
    record
      Pri : Priority;
      Period_In_Milli : Time_Span;
    end record;
  procedure Initialize_Code(S: in out Cyclic_State)
    is abstract;
  procedure Periodic_Code(S: in out Cyclic_State)
    is abstract;
  type Any_Cyclic_State is access all Cyclic_State'Class;

  task type Cyclic (S : Any_Cyclic_State) is
    pragma Priority(S.Pri);
  end Cyclic;
end Cyclics;
```

The body of the package contains the body of the task, which has the standard structure for a periodic task that was introduced in Section 10.2. Note that the calls to `Initialize_Code` and `Periodic_Code` are dispatching operations to the application code associated with the template during instantiation.

```
package body Cyclics is
  task body Cyclic is
    T: Time;
  begin
    S.Initialize_Code;
    T:= Clock + S.Period_In_Milli;
    loop
      delay until T;
      S.Periodic_Code;
      T := T + S.Period_In_Milli;
    end loop;
  end Cyclic;
end Cyclics;
```

In the case of the sporadic task template, two interfaces are used to ensure that the application calls the correct operation. Following the structure given in Section 10.3.1, a task and a protected type is needed:

```
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
package Sporadics is
```

```

type Sporadic_Invoke_Interface is synchronized interface;
procedure Start(S: in out Sporadic_Invoke_Interface)
    is abstract;
type Any_Sporadic_Invoke_Interface is access all
    Sporadic_Invoke_Interface'Class;

type Sporadic_Thread_Interface is synchronized interface;
procedure Wait_Start(S: in out Sporadic_Thread_Interface)
    is abstract;
type Any_Sporadic_Thread_Interface is access all
    Sporadic_Thread_Interface'Class;

type Sporadic_State is abstract tagged
    record
        Pri : Priority;
        Ceiling_Priority : Priority;
        MIT_In_Milli : Time_Span;
    end record;
procedure Initialize_Code(S: in out Sporadic_State)
    is abstract;
procedure Sporadic_Code(S: in out Sporadic_State)
    is abstract;
type Any_Sporadic_State is access all Sporadic_State'Class;

protected type Sporadic_Agent(S: Any_Sporadic_State) is
    new Sporadic_Invoke_Interface and
        Sporadic_Thread_Interface with
        pragma Priority(S.Ceiling_Priority);
    -- for the Start operation
    overriding procedure Start;
    overriding entry Wait_Start;
private
    Start_Open : Boolean := False;
end Sporadic_Agent;

task type Sporadic (S : Any_Sporadic_State;
                    A : Any_Sporadic_Thread_Interface) is
    pragma Priority(S.Pri);
end Sporadic;
end Sporadics;

```

For simplicity the body of the package does not enforce minimum inter-arrival time separation. (A more sophisticated mapping would enable overruns on the interrupting device to be detected and handled – see Section 13.4.1.)

```

package body Sporadics is
    protected body Sporadic_Agent is
        procedure Start is
            begin
                Start_Open := True;
            end Start;

        entry Wait_Start
            when Start_Open is

```



```

begin
    Start_Open := False;
end Wait_Start;
end Sporadic_Agent;

task body Sporadic is
begin
    S.Initialize_Code;
    loop
        A.Wait_Start;
        S.Sporadic_Code;
    end loop;
end Sporadic;
end Sporadics;

```

Finally, the translations assume a package called `Device_Register_Types` that defines the needed control registers for the devices (using the approach given in Chapter 14).

```

with System; use System;
package Device_Register_Types is
    Word : constant := 2;      -- two bytes in a word
    One_Word : constant := 16; -- 16 bits in a word
    -- register field types
    type Device_Error is (Clear, Set);
    type Device_Operation is (Clear, Set);
    type Interrupt_Status is (I_Disabled, I_Enabled);
    type Device_Status is (D_Disabled, D_Enabled);

    -- register type itself
    type Csr is
        record
            Error_Bit   : Device_Error;
            Operation    : Device_Operation;
            Done         : Boolean;
            Interrupt    : Interrupt_Status;
            Device       : Device_Status;
        end record;

    -- bit representation of the register field
    for Device_Error use (Clear => 0, Set => 1);
    for Device_Operation use (Clear => 0, Set => 1);
    for Interrupt_Status use (I_Disabled => 0,
                             I_Enabled => 1);
    for Device_Status use (D_Disabled => 0,
                           D_Enabled => 1);

    for Csr use
        record at mod Word;
            Error_Bit   at 0 range 15 .. 15;
            Operation   at 0 range 10 .. 10;
            Done        at 0 range 7  .. 7;
            Interrupt   at 0 range 6  .. 6;
            Device      at 0 range 0  .. 0;
        end record;
    for Csr'Size use One_Word;
    for Csr'Alignment use Word;
    for Csr'Bit_Order use Low_Order_First;
end Device_Register_Types;

```

15.5.1 The mine control system object

The top-level object in the system is the 'mine control object': it has no operations but does define several types that are used by its child objects. These are, therefore, declared in the following package specification.

```
package Mine_Control_System_Types is
  -- types used in communications between the
  -- Pump_Controller and the Environment_Monitor
  type Methane_Status is (Motor_Safe, Motor_Unsafe);

  -- types used in communications between the
  -- Pump_Controller and the Data_Logger
  type Motor_State_Changes is (Motor_Started,
    Motor_Stopped, Motor_Safe, Motor_Unsafe);
  type Water_Flow is (Yes, No);
  type Water_Mark is (High, Low);

  -- types used in communications between the
  -- Pump_Controller and the Operator_Console
  type Pump_Status is (On, Off);
  type Pump_Condition is (Enabled, Disabled);
  type Operational_Status is record
    Ps : Pump_Status;
    Pc : Pump_Condition;
  end record;
  type Alarm_Reason is (High_Methane, High_Co, No_Air_Flow,
    Ch4_Device_Error, Co_Device_Error,
    Pump_Fault, Pump_Unsafe_To_Operate, Unknown_Error);

  -- types used in communications between the
  -- Environment_Monitor and the Data_Logger
  type Air_Flow_Status is (Air_Flow, No_Air_Flow);
  type Ch4_Reading is new Integer range 0 .. 1023;
  type Co_Reading is new Integer range 0 .. 1023;
  Co_High : constant Co_Reading := 600;
  Ch4_High : constant Ch4_Reading := 400;

  -- there no new types needed for communications between the
  -- Environment_Monitor and the Operator_Console
end Mine_Control_System_Types;
```

Figure 15.3 shows the first level of decomposition of the system. Each of these objects can potentially be implemented on a separate processor. However, for the purpose of this example, a single-processor implementation is considered.

Each of the mine control system children objects is now taken in turn and decomposed.

15.5.2 The pump controller object

The pump controller object is a non-terminal object. It introduces no new types but does provide interfaces illustrated in Figure 15.4; these are specified in the

following package. To allow different views on the pump controller, three interfaces are provided.

```
with Mine_Control_System_Types;
use Mine_Control_System_Types;
package Pump_Controller is
    Pump_Not_Safe : exception; -- raised by Set_Pump

    type Pump_Control is synchronized interface;
    function Request_Status(PC : Pump_Control)
        return Operational_Status is abstract;
    procedure Set_Pump(PC: in out Pump_Control;
        To : Pump_Status) is abstract;
    type Any_Pump_Control is access all Pump_Control'Class;

    type Pump_Inform is synchronized interface;
    procedure Not_Safe(PI: in out Pump_Inform) is abstract;
    procedure Is_Safe (PI: in out Pump_Inform) is abstract;
    type Any_Pump_Inform is access all Pump_Inform'Class;

    type Pump_Operations is synchronized interface and
        Pump_Control and Pump_Inform;
    type Any_Pump_Operations is access all
        Pump_Operations'Class;
end Pump_Controller;
```

As a non-terminal object, objects providing operations for these interfaces must be implemented by child objects. For traceability, each child object of the objects in the top level is viewed as a child package in Ada.

The decomposition appropriate to the pump controller was shown in Figure 15.6. It consists of one non-terminal object (the high-low water sensors object) and two terminal objects (the motor and the water-flow sensor). The figure also showed which pump controller operation was implemented by which child object operation.

It is now possible to give the code for these child objects.

The motor

The real-time attributes for the motor object are given first. As a protected object, only a ceiling priority attribute is needed.

```
package Pump_Controller.Motor_Rtatt is
    Ceiling_Priority: constant := 10;
end Pump_Controller.Motor_Rtatt;
```

The package specification for the implementation of the motor object is given next. It defines a protected type that implements all the interfaces provided by the pump controller. In this chapter, all protected objects generated to implement synchronization constraints are tagged as Agent. The type is parameterized so that (if necessary) multiple motors can be defined. An access reference to the device's control register is provided, along with references to the interfaces that allow the agent to communicate with other objects in the system, and its ceiling priority.

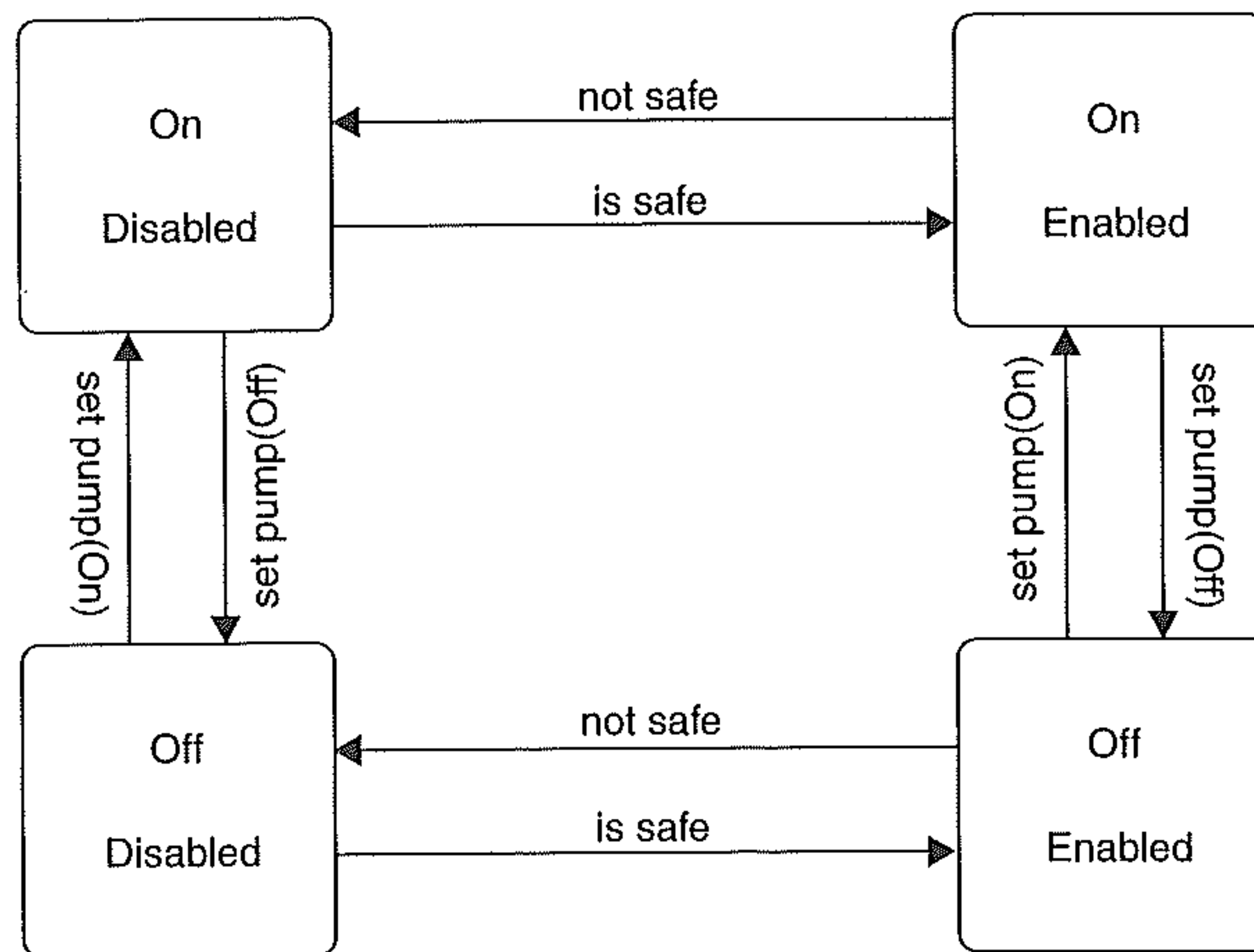


Figure 15.9 State transition diagram for the motor.

```

with Device_Register_Types; use Device_Register_Types;
with Environment_Monitor; use Environment_Monitor;
with Data_logger; use Data_logger;
with System; use System;
package Pump_Controller.Motor is -- PROTECTED
  protected type Motor_Agent(Pcsr : access Csr;
    Safety : Any_Methane_Check_Safety;
    Log : Any_Pump_Log;
    Ceiling: Priority) is new Pump_Operations with
    pragma Priority(Ceiling);
    overriding function Request_Status return Operational_Status;
    overriding procedure Set_Pump(To : Pump_Status);
    overriding procedure Not_Safe;
    overriding procedure Is_Safe;
  private
    Motor_Status : Pump_Status := Off;
    Motor_Condition : Pump_Condition := Disabled;
  end Motor_Agent;
end Pump_Controller.Motor;

```

The state of the motor is defined by two variables, one which indicates whether the pump should be on or off, and the other whether the pump is enabled or disabled. The pump is disabled when it is unsafe to operate. A state transition diagram for the motor is shown in Figure 15.9. Only in the 'On-Enabled' state will the pump actually be operating.

The body of the `Motor` package contains the body of the `Motor_Agent` protected type, which implements the state transitions and their associated actions.

```

with Data_Logger;
with Environment_Monitor; use Environment_Monitor;
with System; use System;
with System.Storage_Elements; use System.Storage_Elements;

```



```

package body Pump_Controller.Motor is

protected body Motor_Agent is
  procedure Not_Safe is
  begin
    if Motor_Status = On then
      Pcsr.Operation := Clear; -- turn off motor
      Log.Motor_Log(Motor_Stopped);
    end if;
    Motor_Condition := Disabled;
    Log.Motor_Log(Motor_Unsafe);
  end Not_Safe;

  procedure Is_Safe is
  begin
    if Motor_Status = On then
      Pcsr.Operation := Set; -- start motor
      Log.Motor_Log(Motor_Started);
    end if;
    Motor_Condition := Enabled;
    Log.Motor_Log(Motor_Safe);
  end Is_Safe;

  function Request_Status return Operational_Status is
  begin
    return (Ps => Motor_Status, Pc => Motor_Condition);
  end Request_Status;

  procedure Set_Pump(To : Pump_Status) is
  begin
    if To = On then
      if Motor_Status = Off then
        if Motor_Condition = Disabled then
          raise Pump_Not_Safe;
        end if;
        if Safety.Check_Safe = Motor_Safe then
          Motor_Status := On;
          Pcsr.Operation := Set; -- turn on motor
          Log.Motor_Log(Motor_Started);
        else
          raise Pump_Not_Safe;
        end if;
      end if;
    else
      if Motor_Status = On then
        Motor_Status := Off;
        if Motor_Condition = Enabled then
          Pcsr.Operation := Clear; -- turn off motor
          Log.Motor_Log(Motor_Stopped);
        end if;
      end if;
    end if;
  end Set_Pump;
end Motor_Agent;
end Pump_Controller.Motor;

```

Water-flow sensor handling object

The water-flow sensor is a cyclic object, and therefore has the following real-time attributes:

```
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
package Pump_Controller.Water_Flow_Sensor_Rtatt is
  Period : Time_Span := Milliseconds(1000);
  Thread_Priority : constant Priority := 9;
end Pump_Controller.Water_Flow_Sensor_Rtatt;
```

In order to instantiate the periodic task template, it is first necessary to extend the `Cyclic_State` to contain the state needed for the sensor, and to override the operations. The actual sensor object will be created when the architecture itself is instantiated.

```
with Cyclics; use Cyclics;
with Device_Register_Types; use Device_Register_Types;
with Operator_Console; use Operator_Console;
with Data_Logger; use Data_Logger;
package Pump_Controller.Water_Flow_Sensor is -- CYCLIC
  -- calls Operator_Console.Alarm
  -- calls Data_Logger.Water_Flow_Log
  -- calls Motor.Request_Status

  type Sensor_State(
    Wfcsr : access Csr; Motor: Any_Pump_Control;
    Operator : Any_Alarms;
    Logger : Any_Pump_Log) is new Cyclic_State with
  record
    Flow : Water_Flow := No;
    Current_Pump_Status,
    Last_Pump_Status : Pump_Status := Off;
  end record;
  overriding procedure Initialize_Code(
    S : in out Sensor_State);
  overriding procedure Periodic_Code(
    S : in out Sensor_State);
end Pump_Controller.Water_Flow_Sensor;
```

The body contains two subprograms: one for initializing the sensor (`Initialize`), and the other for the code to be executed each period (`Periodic_Code`). Every invocation, the task simply checks that if the pump is on, water is flowing and if the pump is off, no water flows. Alarms are sounded if these two invariants are violated.

```
with Pump_Controller.Motor; use Pump_Controller.Motor;
with Mine_Control_System_Types;
use Mine_Control_System_Types;
package body Pump_Controller.Water_Flow_Sensor is
  procedure Initialize_Code(S : in out Sensor_State) is
  begin
    -- enable device
    S.Wfcsr.Device := D_Enabled;
  end Initialize_Code;
```



```

procedure Periodic_Code(S : in out Sensor_State) is
begin
  S.Current_Pump_Status := S.Motor.Request_Status.Ps;
  if (S.Wfcsr.Operation = Set) then
    S.Flow := Yes;
  else
    S.Flow := No;
  end if;
  if S.Current_Pump_Status = On and
    S.Last_Pump_Status = On and S.Flow = No then
    S.Operator.Sound_Alarm(Pump_Fault);
  elsif S.Current_Pump_Status = Off and
    S.Last_Pump_Status = Off and S.Flow = Yes then
    S.Operator.Sound_Alarm(Pump_Fault);
  end if;
  S.Last_Pump_Status := S.Current_Pump_Status;
  S.Logger.Water_Flow_Log(S.Flow);
end Periodic_Code;
end Pump_Controller.Water_Flow_Sensor;

```

The high-low water sensors object

This object is an active (non-terminal) object. Both of its provided operations are called from interrupts, and are therefore defined to be operations in a protected type. As the interrupts are mutually exclusive, the same protected object is used for both interrupts.

```

with System; use System;
with Ada.Interrupts; use Ada.Interrupts;
with Sporadics; use Sporadics;
package Pump_Controller.High_Low_Water_Sensors is
  -- Operations called from interrupt handlers
  -- No public interface

  type High_Sensor is protected interface;
  procedure Sensor_High_Ih(HS : in out High_Sensor)
    is abstract;

  type Low_Sensor is protected interface;
  procedure Sensor_Low_Ih(LS : in out Low_Sensor)
    is abstract;

  protected type HLW_Agent(Waterh_Interrupt: Interrupt_Id;
    Waterl_Interrupt: Interrupt_Id;
    HW : Any_Sporadic_Invoke_Interface;
    LW : Any_Sporadic_Invoke_Interface;
    Ceiling : Priority)
    is new High_Sensor and Low_Sensor with
    pragma Priority(Ceiling);
    overriding procedure Sensor_High_Ih;
    pragma Attach_Handler(Sensor_High_Ih, Waterh_Interrupt);
    -- assigns interrupt handler
    overriding procedure Sensor_Low_Ih;
    pragma Attach_Handler(Sensor_Low_Ih, Waterl_Interrupt);
    -- assigns interrupt handler
  end HLW_Agent;
end Pump_Controller.High_Low_Water_Sensors;

```

The decomposition of the high–low water sensors was given in Figure 15.7. It consists of two sporadic objects: one to handle the high water interrupt and the other to handle the low water interrupt. Hence, the body of the high–low water sensors object simply calls the start method to release the tasks.

```
package body Pump_Controller.High_Low_Water_Sensors is
  protected body HLW_Agent is
    procedure Sensor_High_Ih is
    begin
      HW.Start;
    end Sensor_High_Ih;

    procedure Sensor_Low_Ih is
    begin
      LW.Start;
    end Sensor_Low_Ih;
  end HLW_Agent;
end Pump_Controller.High_Low_Water_Sensors;
```

The high water handler and the low water handler objects

The two sporadic objects responsible for responding to the high and low water interrupts are considered next. The generated code for the two objects is identical in structure. Here only the high water handler is given. First its real-time attributes are defined:

```
with System; use System;
package Pump_Controller.High_Low_Water_Sensors.
  Hw_Handler_Rtatt is
  Ceiling_Priority : constant Priority := Priority'Last;
  Thread_Priority : constant Priority := 6;
end Pump_Controller.High_Low_Water_Sensors.Hw_Handler_Rtatt;
```

Note that as the sporadic object will be invoked from a interrupt handler, it must have the maximum ceiling priority. The code for the object's operations is straightforward, and given below.

```
with Sporadics; use Sporadics;
with Device_Register_Types; use Device_Register_Types;
with Pump_Controller; use Pump_Controller;
with Data_logger; use Data_Logger;
with Operator_Console; use Operator_Console;
package Pump_Controller.High_Low_Water_Sensors.Hw_Handler is
  -- SPORADIC
  type HW_Sensor_State(
    Hwcsr : access Csr; Operator : Any_Alarms;
    Motor: Any_Pump_Control; Logger : Any_Pump_Log)
    is new Sporadic_State with null record;
  overriding procedure Initialize_Code(
    S : in out HW_Sensor_State);
  overriding procedure Sporadic_Code(
    S : in out HW_Sensor_State);
end Pump_Controller.High_Low_Water_Sensors.Hw_Handler;
```



```

package body Pump_Controller.High_Low_Water_Sensors.
    Hw_Handler is
  procedure Sporadic_Code(S : in out HW_Sensor_State) is
  begin
    S.Motor.Set_Pump(On);
    S.Logger.High_Low_Water_Log(High);
    S.Hwcsr.Interrupt := I_Disabled;
  exception
    when Pump_Not_Safe =>
      S.Operator.Sound_Alarm(Pump_Unsafe_To_Operate);
  end Sporadic_Code;

  procedure Initialize_Code(S : in out HW_Sensor_State) is
  begin
    S.Hwcsr.Device := D_Enabled;
    S.Hwcsr.Interrupt := I_Enabled;
  end Initialize_Code;
end Pump_Controller.High_Low_Water_Sensors.Hw_Handler;

```

The low water handler object code is similarly structured.

15.5.3 Environment monitoring

The environment monitor subsystem's goal is to monitor the mine to ensure that it is safe for the workforce. It is an active object with two interfaces: one is used by the pump controller to determine if the methane level is currently safe or not; the other is used by the internal objects to set the status.

```

with Mine_Control_System_Types;
use Mine_Control_System_Types;
package Environment_Monitor is
  type Methane_Safety_Check is synchronized interface;
  function Check_Safe(S : in Methane_Safety_Check)
    return Methane_Status is abstract;
  type Any_Methane_Safety_Check is access all
    Methane_Safety_Check'Class;

  type Check_Safety is synchronized interface and
    Methane_Safety_Check;
  procedure Set_Safety(S: in out Check_Safety;
    M : Methane_Status) is abstract;
  type Any_Check_Safety is access all Check_Safety'Class;
end Environment_Monitor;

```

The decomposition of the environment monitor subsystem was shown in Figure 15.8. Note that the Check_Safe subprogram allows the pump controller to observe the current state of the methane level without blocking, via the CH₄ status protected object. All other components are periodic activities.

The following subsections illustrate the decomposition. The real-time attributes are not shown.

CH₄ status object

The CH₄ status object simply contains data which indicates whether it is safe to operate the pump. The code for the operations is straightforward.

```
with Mine_Control_System_Types;
use Mine_Control_System_Types;
with System; use System;
package Environment_Monitor.Ch4_Status is -- PROTECTED
    protected type Methane_Agent(Ceiling : Priority)
        is new Check_Safety with
        pragma Priority(Ceiling);
        overriding function Check_Safe return Methane_Status;
        overriding procedure Set_Safety(M : Methane_Status);
    private
        Current_Status : Methane_Status := Motor_Unsafe;
    end Methane_Agent;
end Environment_Monitor.Ch4_Status;

package body Environment_Monitor.Ch4_Status is
    protected body Methane_Agent is
        function Check_Safe return Methane_Status is
        begin
            return Current_Status;
        end Check_Safe;

        procedure Set_Safety(M : Methane_Status) is
        begin
            Current_Status := M;
        end Set_Safety;
    end Methane_Agent;
end Environment_Monitor.Ch4_Status;
```

CH₄ sensor handling object

The function of the CH₄ sensor is to measure the level of methane in the environment. As with the previous water-flow sensor object, the approach is to extend the *Cyclic_State* type and override the operations.

```
with Cyclics; use Cyclics;
with Device_Register_Types; use Device_Register_Types;
with Operator_Console; use Operator_Console;
with Data_Logger; use Data_Logger;
with Pump_Controller; use Pump_Controller;
package Environment_Monitor.Ch4_Sensor is -- CYCLIC
    -- calls Pump_Controller.Is_Safe
    -- calls Pump_Controller.Not_Safe
    -- calls Operator_Console.Alarm
    -- calls Data_Logger.Ch4_Log

    type CH4_Sensor_State(
        Ch4csr : access Csr;
        Ch4dbr : access Ch4_Reading;
        CH4_Status : Any_CH4_Safety;
```



```

    Motor: Any_Pump_Inform;
    Operator : Any_Alarms;
    Logger : Any_Environment_Log)
    is new Cyclic_State with
record
    Ch4_Present : Ch4_Reading;
end record;
overriding procedure Initialize_Code(
    S : in out CH4_Sensor_State);
overriding procedure Periodic_Code(
    S : in out CH4_Sensor_State);
end Environment_Monitor.Ch4_Sensor;

```

The requirement is that the methane level should not rise above a threshold. Inevitably, around this threshold the sensor will continually signal safe and unsafe. To avoid this jitter, lower and upper bounds on the threshold are used. Note that, as the ADC takes some time to produce its result, the conversion is requested at the end of one period to be used at the start of the next.

```

package body Environment_Monitor.Ch4_Sensor is
    Jitter_Range : constant Ch4_Reading := 40;

    procedure Initialize_Code(S : in out Ch4_Sensor_State) is
    begin
        -- enable device
        S.Ch4csr.Device := D_Enabled;
        S.Ch4csr.Operation := Set;
    end Initialize_Code;

    procedure Periodic_Code(S : in out Ch4_Sensor_State) is
        Methane : Methane_Status;
    begin
        if not S.Ch4csr.Done then
            S.Operator.Sound_Alarm(Ch4_Device_Error);
        else
            -- read device register for sensor value
            S.Ch4_Present := S.Ch4dbr.all;
            Methane := S.CH4_Status.Check_Safe;
            if S.Ch4_Present > Ch4_High then
                if Methane = Motor_Safe then
                    S.Motor.Not_Safe;
                    S.Ch4_Status.Set_Safety(Motor_Unsafe);
                    S.Operator.Sound_Alarm(High_Methane);
                end if;
            elsif (S.Ch4_Present < (Ch4_High - Jitter_Range)) and
                (Methane = Motor_Unsafe) then
                S.Motor.Is_Safe;
                S.Ch4_Status.Set_Safety(Motor_Safe);
            end if;
            S.Logger.Ch4_Log(S.Ch4_Present);
        end if;
        S.Ch4csr.Operation := Set;
        -- start conversion for next iteration
    end Periodic_Code;
end Environment_Monitor.Ch4_Sensor;

```

15.5.4 Air-flow sensor handling object

The air-flow sensor is another periodic object which simply monitors the flow of air in the mine.

```

with Cyclics; use Cyclics;
with Device_Register_Types; use Device_Register_Types;
with Operator_Console; use Operator_Console;
with Data_Logger; use Data_Logger;
package Environment_Monitor.Air_Flow_Sensor is -- CYCLIC
  -- calls Data_Logger.Air_Flow_Log
  -- calls Operator_Console.Alarm
  type AF_Sensor_State(Afcsr : access Csr;
    Operator : Any_Alarms; Logger : Any_Environment_Log)
    is new Cyclic_State with
  record
    Air_Flow_Reading : Boolean := True;
  end record;
  overriding procedure Initialize_Code(S : in out AF_Sensor_State);
  overriding procedure Periodic_Code(S : in out AF_Sensor_State);
end Environment_Monitor.Air_Flow_Sensor;

package body Environment_Monitor.Air_Flow_Sensor is
  procedure Initialize_Code(S : in out AF_Sensor_State) is
  begin
    -- enable device
    S.Afcsr.Device := D_Enabled;
  end Initialize_Code;

  procedure Periodic_Code(S : in out AF_Sensor_State) is
  begin
    -- read device register for flow indication
    -- (operation bit set to 1);
    S.Air_Flow_Reading := S.Afcsr.Operation = Set;
    if not S.Air_Flow_Reading then
      S.Operator.Sound_Alarm(No_Air_Flow);
      S.Logger.Air_Flow_Log(No_Air_Flow);
    else
      S.Logger.Air_Flow_Log(Air_Flow);
    end if;
  end Periodic_Code;
end Environment_Monitor.Air_Flow_Sensor;

```

15.5.5 CO sensor handling object

The CO sensor is, again, straightforward in its implementation.

```

with Cyclics; use Cyclics;
with Device_Register_Types; use Device_Register_Types;
with Operator_Console; use Operator_Console;
with Data_Logger; use Data_Logger;
package Environment_Monitor.Co_Sensor is -- CYCLIC
  -- calls Data_Logger.Co_log
  -- calls Operator_Console.Alarm

```



```

type CO_Sensor_State(Cocsr : access Csr;
    Codbr : access CO_Reading; Operator : Any_Alarms;
    Logger : Any_Environment_Log) is
    new Cyclic_State with
record
    Co_Present : Co_Reading;
end record;
overriding procedure Initialize_Code(S : in out CO_Sensor_State);
overriding procedure Periodic_Code(S : in out CO_Sensor_State);
end Environment_Monitor.Co_Sensor;

package body Environment_Monitor.Co_Sensor is
procedure Initialize_Code(S : in out CO_Sensor_State) is
begin
    -- enable device
    S.Cocsr.Device := D_Enabled;
    S.Cocsr.Operation := Set; -- start conversion
end Initialize_Code;

procedure Periodic_Code(S : in out CO_Sensor_State) is
begin
    if not S.Cocsr.Done then
        S.Operator.Sound_Alarm(Co_Device_Error);
    else
        -- read device register for sensor value
        S.Co_Present := S.Codbr.all;
        if S.Co_Present > Co_High then
            S.Operator.Sound_Alarm(High_Co);
        end if;
        S.Logger.Co_Log(S.Co_Present);
    end if;
    S.Cocsr.Operation := Set; -- start conversion
end Periodic_Code;
end Environment_Monitor.Co_Sensor;

```

15.5.6 Data logger

Only the interface to the data logger object is shown.

```

with Mine_Control_System_Types;
use Mine_Control_System_Types;
package Data_Logger is -- ACTIVE
    type Pump_Log is interface;
    procedure High_Low_Water_Log(Pump: in out Pump_Log;
        Mark : Water_Mark) is abstract;
    procedure Water_Flow_Log(Pump: in out Pump_Log;
        Reading : Water_Flow) is abstract;
    procedure Motor_Log(Pump: in out Pump_Log;
        State : Motor_State_Changes) is abstract;
    type Any_Pump_Log is access all Pump_Log'Class;

    type Environment_Log is interface;
    procedure Co_Log(Envir: in out Environment_Log;
        Reading : Co_Reading) is abstract;

```

```

procedure Ch4_Log(Envir: in out Environment_Log;
                  Reading : Ch4_Reading) is abstract;
procedure Air_Flow_Log(Envir: in out Environment_Log;
                      Reading : Air_Flow_Status) is abstract;
type Any_Environment_Log is access all Environment_Log'Class;

type Status_Log is interface and Pump_Log
    and Environment_Log;

type Logger is new Status_log with null record;
procedure High_Low_Water_Log(Pump: in out Logger;
                             Mark : Water_Mark);
procedure Water_Flow_Log(Pump: in out Logger;
                          Reading : Water_Flow);
procedure Motor_Log(Pump: in out Logger;
                     State : Motor_State_Changes);
procedure Co_Log(Envir: in out Logger;
                  Reading : Co_Reading);
procedure Ch4_Log(Envir: in out Logger;
                  Reading : Ch4_Reading);
procedure Air_Flow_Log(Envir: in out Logger;
                       Reading : Air_Flow_Status);
end Data_Logger;

```

15.5.7 Operator console

Only the interface to the operator console object is shown.

```

with Mine_Control_System_Types;
use Mine_Control_System_Types;
package Operator_Console is -- ACTIVE
    type Alarms is interface;
    procedure Sound_Alarm(A: Alarms; Reason : Alarm_Reason)
        is abstract;
    type Any_Alarms is access all Alarms'Class;

    type Console is new Alarms with null record;
    procedure Sound_Alarm(C: Console; Reason : Alarm_Reason);
end Operator_Console;

```

15.5.8 Configuring the system

Having defined all the types representing the needed objects, it is now possible to consider the instances to instantiate the software architecture. The configuration is illustrated below:

```

-- with and use clauses omitted
package Mine_Control_System is
    -- The Data_Logger
    The_Logger : aliased Logger;

    -- The Operators Console

```



```

The_Console : aliased Console;

-- CH4 status
CH4_Status_Agent : aliased Methane_Agent (
    CH4_Status_Rtatt.Ceiling_Priority);

-- The Motor
Ctrl_Reg_Addr1 : constant Address := To_Address(16#Aa14#);
Pcsr : aliased Device_Register_Types.Csr :=
    (Error_Bit => Clear, Operation => Set,
     Done => False, Interrupt => I_Enabled,
     Device => D_Enabled);
for Pcsr'Address use Ctrl_Reg_Addr1;

Pump_Motor : aliased Pump_Controller.Motor.Motor_Agent(Pcsr'Access,
    CH4_Status_Agent'Access,
    The_Logger'Access, Motor_Rtatt.Ceiling_Priority);

-- The Waterflow Sensor
Ctrl_Reg_Addr2 : constant Address := To_Address(16#Aa14#);
Wfcsr : aliased Device_Register_Types.Csr;
for Wfcsr'Address use Ctrl_Reg_Addr2;
Water_Flow_Sensor_State: aliased Pump_Controller.
    Water_Flow_Sensor.Sensor_State := (
        Wfcsr => Wfcsr'Access,
        Motor => Pump_Motor'Access,
        Operator => The_Console'Access,
        Logger => The_Logger'Access,
        Flow => No,
        Current_Pump_Status => Off,
        Last_Pump_Status => Off,
        Period_In_Milli => Water_Flow_Sensor_Rtatt.Period,
        Pri'=> Water_Flow_Sensor_Rtatt.Thread_Priority);
Water_Flow_Thread : Cyclic(Water_Flow_Sensor_State'Access);

-- HW Handler
Hw_Cntrl_Reg_Addr : constant Address :=
    To_Address(16#Aa10#);
Hwcsr : aliased Device_Register_Types.Csr;
for Hwcsr'Address use Hw_Cntrl_Reg_Addr;
High_Water_State : aliased Pump_Controller.
High_Low_Water_Sensors.HW_Handler.HW_Sensor_State(
    Hwcsr'Access,
    The_Console'Access,
    Pump_Motor'Access,
    The_Logger'Access);
HW_Agent : aliased Sporadic_Agent(High_Water_State'Access);
HW_Thread : Sporadic(High_Water_State'Access,
    HW_Agent'Access);

-- LW Handler
Lw_Cntrl_Reg_Addr : constant Address :=
    To_Address(16#Aa12#);
Lwcsr : aliased Device_Register_Types.Csr;
for Hwcsr'Address use Hw_Cntrl_Reg_Addr;
Low_Water_State : aliased Pump_Controller.

```

```

    High_Low_Water_Sensors.LW_Handler.LW_Sensor_State(
        Lwcsr'Access,
        The_Console'Access,
        Pump_Motor'Access,
        The_Logger'Access);
    LW_Agent : aliased Sporadic_Agent(Low_Water_State'Access);
    LW_Thread : Sporadic(Low_Water_State'Access,
        LW_Agent'Access);

-- The High Low Water Sensor
    HLW_Interrupt_Handler : Agent(Ada.Interrupts.Names.HW,
        Ada.Interrupts.Names.LW,
        HW_Agent'Access, LW_Agent'Access,
        High_Low_Water_Sensors_Rtatt.Ceiling_Priority);

-- CO Sensor, air flow sensor, and CH4 sensor
-- are similar to airflow sensor

end Mine_Control_System;
```

15.6 Translation to Real-Time Java

In the translation to Real-Time Java, each active object is translated into a Java package, and each terminal object to Java classes in that package. Here only parts of the mapping are illustrated.

15.6.1 The mine control system object

The top-level object is an active object. The types used by its child objects are declared as classes in the associated directory. For example:

```

package mineControlSystem;
public enum MethaneStatus{MOTOR_SAFE, MOTOR_UNSAFE}

package mineControlSystem;
public enum MotorStateChange {
    MOTOR_STARTED, MOTOR_STOPPED, MOTOR_SAFE, MOTOR_UNSAFE}

package mineControlSystem;

public class Ch4Reading {

    public Ch4Reading(double reading) {
        this.reading = reading;
    }

    public boolean Ch4High() {
        if (reading > CH4_HIGH) return true;
        else return false;
    }
    private double reading;
```



```

    private static final double CH4_HIGH;
}

package mineControlSystem;
public enum AlarmReason {PUMP_FAULT, PUMP_UNSAFE_TO_OPERATE,
                        NO_AIRFLOW, AIRFLOW_DEVICE_ERROR,
                        CO_PRESENT, CO_DEVICE_ERROR,
                        CH4_PRESENT, CH4_DEVICE_ERROR}

```

For each of the mine control system's child active objects there are directories containing the associated code. Here, only the pump controller is shown.

15.6.2 The pump controller object

The pump controller object is an active object with several interfaces defined. These are shown below:

```

package mineControlSystem.pumpController;
import mineControlSystem.*;
public interface PumpControl {
    public OperationalStatus requestStatus();
    public void setPump(PumpStatus to) throws PumpNotSafe;
}

package mineControlSystem.pumpController;
import mineControlSystem.*;
public interface PumpInform {
    public void notSafe();
    public void isSafe();
}

package mineControlSystem.pumpController;
import mineControlSystem.*;
public interface PumpOperations extends
    PumpControl, PumpInform {}

```

The pump controller has two terminal child objects and one active one. The active one consists of two further terminal objects. Here, one protected, one cyclic and one sporadic object are taken to illustrate the translation.

The motor

The real-time attributes for the protected motor object are given first. For simplicity, only the ceiling priority attribute is shown.

```

package mineControlSystem.pumpController;
public class MotorRtatt {
    public static final int CeilingPriority = 10;
}

```

The motor object is:

```

package mineControlSystem.pumpController;

import mineControlSystem.dataLogger.PumpLog;
import deviceRegister.DeviceRegister;
import mineControlSystem.*;

public class Motor implements PumpOperations {
    public Motor(long deviceAddress, PumpLog dl) {
        motorStatus = PumpStatus.OFF;
        motorCondition = PumpCondition.DISABLED;
        myDevice = new DeviceRegister(deviceAddress);
        myDl = dl;
    }

    private PumpStatus motorStatus;
    private PumpCondition motorCondition;
    private DeviceRegister myDevice;
    private PumpLog myDl;

    public synchronized OperationalStatus requestStatus() {
        OperationalStatus ps = new OperationalStatus();
        ps.motorStatus = PumpStatus.ON;
        ps.pumpCondition = PumpCondition.ENABLED;
        return ps;
    }

    public synchronized void notSafe() {
        if(motorStatus == PumpStatus.ON) {
            myDevice.clearOperate();
            myDl.motorLog(MotorStateChange.MOTOR_STOPPED);
        }
        motorCondition = PumpCondition.DISABLED;
        myDl.motorLog(MotorStateChange.MOTOR_UNSAFE);
    }

    public synchronized void isSafe() {
        if(motorStatus == PumpStatus.ON) {
            myDevice.setOperate();
            myDl.motorLog(MotorStateChange.MOTOR_STARTED);
        }
        motorCondition = PumpCondition.ENABLED;
        myDl.motorLog(MotorStateChange.MOTOR_SAFE);
    }

    public synchronized void setPump(PumpStatus to) throws PumpNotSafe {
        if (to == PumpStatus.ON) {
            if (motorStatus == PumpStatus.OFF) {
                if(motorCondition == PumpCondition.DISABLED) {
                    throw new PumpNotSafe();
                } else{
                    // turn on pump
                    motorStatus = PumpStatus.ON;
                    myDevice.setOperate();
                    myDl.motorLog(MotorStateChange.MOTOR_STARTED);
                }
            }
        }
    }
}

```



```

    }
  }
} else {
  if(motorStatus == PumpStatus.ON) {
    motorStatus = PumpStatus.OFF;
    if(motorCondition == PumpCondition.ENABLED) {
      myDevice.clearOperate();
      myDl.motorLog(MotorStateChange.MOTOR_STOPPED);
    }
  }
}
}
}
}

```

The state of the motor is again defined by two variables: one which indicates whether the pump should be on or off, and the other whether the pump is enabled or disabled. The pump is disabled when it is unsafe to operate. The type `Motor_State_Changes` is used to indicate state changes to the data logger.

The code assumes that devices register types are declared in a package called `deviceRegister`.

```

package deviceRegister;
import javax.realtime.*;
public class DeviceRegister
{
  public DeviceRegister(long base) {
    try {
      rawMemory = new RawMemoryAccess(IO_Page, base, REG_SIZE);
    }
    catch(Exception e) { throw new IllegalArgumentException();}
  }

  RawMemoryAccess rawMemory;
  short shadow;
  final long REG_SIZE = 2;
  final short ENABLE_DEVICE = 01;
  final short SET_OPERATION = 02000;

  public void enableDevice() {
    try {
      shadow = ENABLE_DEVICE;
      rawMemory.setShort(0, shadow);
    } catch(Exception e) {};
  }

  public void setOperate() {
    try {
      shadow = ENABLE_DEVICE | SET_OPERATION;
      rawMemory.setShort(0, shadow);
    } catch(Exception e) {};
  }

  public void clearOperate() {
    try {

```

```

        shadow = ENABLE_DEVICE | SET_OPERATION;
        rawMemory.setShort(0, shadow);
    } catch(Exception e) {};
}

public boolean isSet() {
    try {
        shadow = rawMemory.getShort(0);
        if ((shadow & SET_OPERATION) != 0) return true;
        else return false;
    } catch(Exception e) {return false;}
}
}

```

Water-flow sensor handling object

The water-flow sensor is a cyclic object, and therefore has the following real-time attributes.

```

package mineControlSystem.pumpController;
import javax.realtime.*;
public class WaterFlowSensorRtatt {
    public static final RelativeTime period =
        new RelativeTime(1000,0);
    public static final RelativeTime deadline =
        new RelativeTime(1000,0);
    public static final int priority = 9;
}

package mineControlSystem.pumpController;

import javax.realtime.*;
import mineControlSystem.dataLogger.PumpLog;
import mineControlSystem.operatorConsole.Alarms;
import mineControlSystem.*;
import deviceRegister.*;

public class WaterFlowSensor extends RealtimeThread {
    public WaterFlowSensor(PumpOperations pump, PumpLog logger,
        Alarms alarm) {
        super(new PriorityParameters(WaterFlowSensorRtatt.priority),
            new PeriodicParameters(WaterFlowSensorRtatt.period));
        myPump = pump;
        myLog = logger;
        myAlarm = alarm;
        try {
            myReg = new DeviceRegister(CsrBaseAddress);
            myReg.enableDevice();
        } catch(Exception e) {
            throw new IllegalArgumentException();
        }
    }
}

private final long CsrBaseAddress = 01400;

```



```

private final short channel = 6;
private final long REG_SIZE = 2;
private DeviceRegister myReg;

private WaterFlow flow = WaterFlow.NO;
private PumpStatus currentPumpStatus = PumpStatus.OFF;
private PumpStatus lastPumpStatus = PumpStatus.OFF;
private OperationalStatus status;
private PumpOperations myPump;
private PumpLog myLog;
private Alarms myAlarm;

public void periodicCode() {
    status = myPump.requestStatus();
    if (myReg.isSet()) {
        currentPumpStatus = PumpStatus.ON;
    } else currentPumpStatus = PumpStatus.OFF;

    if (currentPumpStatus == PumpStatus.ON &&
        lastPumpStatus == PumpStatus.ON &&
        flow == WaterFlow.YES) {
        myAlarm.soundAlarm(AlarmReason.PUMP_FAULT);
    } else {
        if (currentPumpStatus == PumpStatus.OFF &&
            lastPumpStatus == PumpStatus.OFF &&
            flow == WaterFlow.YES) {
            myAlarm.soundAlarm(AlarmReason.PUMP_FAULT);
        }
    }
    myLog.waterFlowLog(flow);
    lastPumpStatus = currentPumpStatus;
}

public void run() {
    while(true) {
        periodicCode();
        waitForNextPeriod();
    }
}
}

```

The high–low water sensor objects

The two interrupts from the high and low water sensors are treated in exactly the same way. Each is represented by a Real-Time Java ‘happening’ and consequently there is an associated bound asynchronous event handler. An example of the high water interrupt handler is given below.

```

package mineControlSystem.pumpController.highLowWaterSensors;
import javax.realtime.*;
import mineControlSystem.*;
import mineControlSystem.pumpController.*;

public class HWInterruptHandler extends BoundAsyncEventHandler {

```

```

public HWInterruptHandler(PumpControl pump, Alarms alarm) {
    super(new PriorityParameters(HWInterruptHandlerRtatt.priority),
          new SporadicParameters(HWInterruptHandlerRtatt.MIT),
          null, null, null, false, null);
    myPump = pump;
    myAlarms = alarms
    // set device status etc
}

private PumpControl myPump;
private Alarms myAlarms

public void handleAsyncEvent()
{
    try {
        myPump.setPump(PumpStatus.ON);
    } catch (PumpNotSafe E) {
        myAlarms.soundAlarm(AlarmReason.PUMP_UNSAFE_TO_OPERATE);
    }
    // set device status etc
}
}

```

Although not all the Java code has been given, the reader should be able to complete the study.

15.7 Fault tolerance and distribution

Chapter 2 identified four sources of faults which can result in an embedded system software failure.

- (1) Inadequate specification.
- (2) Faults introduced from design errors in software components.
- (3) Faults introduced by failure of one or more processor components of the embedded system.
- (4) Faults introduced by transient or permanent interference in the supporting communication subsystem.

It is these last three on which this book has concentrated. They are now discussed in turn in relation to the case study. Ada's and Real-Time Java's approach to software fault tolerance is to use exception handling as a framework from which error recovery can be built.

15.7.1 Design errors

As this case study is necessarily simplified, the scope for fault tolerance of software design errors is small. In the example, the HRT-HOOD design methodology, in conjunction with the language's data abstraction facilities, has been used in an attempt to prevent

faults from entering the system during the design and implementation phases. In a real application, this would then be followed by a comprehensive testing phase to remove any faults that had, nevertheless, been introduced. Simulations and model checking techniques may also be used.

Any residual design faults in the program will cause unanticipated errors to occur. Although backward error recovery or *N*-version programming is ideal for recovering from these types of errors, there is little scope in the example for design diversity. Although a two version system, one in Ada and the other in Real-Time Java, could be used.

If within the case study it is assumed that all unanticipated errors result in exceptions being raised, each operation could be protected by a 'catch all' exception handler. For example, the CH₄ sensor object could be modified to inform the operator if an unexpected error occurs. For safety, in this situation, an attempt is made to turn off the motor. In Ada, this would take the following form:

```

procedure Periodic_Code(S : in out Sensor_State) is
begin
    ...
exception
    when others =>
        S.Operator.Sound_Alarm(Unknown_Error);
        S.Ch4_Status.Set_Safety(Motor_Unsafe);
        S.Motor.Not_Safe;
end Periodic_Code;
    
```

Although mine flooding is serious, the application's requirements dictate that fire is more dangerous; therefore error handling always attempts to ensure that the pump is turned off (fail safe).

It should be noted that all interactions which manipulate the pump and the status of methane should be in the form of atomic actions. The code given in the case study allows another task to determine that the motor is in an unsafe position, even though the methane status might indicate that the pump is safe to operate. This might lead to a race condition (see Exercise 15.2).

15.7.2 Processor and communication failure

In general, if the mine control system was implemented on a single-processor computer and any part of the processor failed then the whole system would be in jeopardy. Consequently, either some form of hardware redundancy must be applied or distribution is required. Control systems of the kind found in mines are naturally distributed. The top-level decomposition illustrated in Figure 15.3 shows four components that could clearly execute on distinct processes (see Exercise 15.3).

Neither Ada nor Java defines failure semantics for partially failed programs. However, in both languages, an exception is raised by the underlying implementation when it is unable to make contact with a remote node (see Section 6.8).

It has been assumed that all transient communication failures will be masked by the underlying distributed system's implementation. If a more permanent failure occurs then an appropriate exception should be raised by the implementation, which the application

can then handle. For example, if the remote call to `Is_Safe` generates an exception, the pump should be disabled.

15.7.3 Other hardware failures

The above assumes that only the processor and the communications subsystem can fail. Clearly, it is equally likely that the sensors may fail either through deterioration or through damage. In the example presented in this chapter, no attempt has been made to increase the reliability of the sensors as this book has only touched upon hardware redundancy techniques. One approach would be to replicate each sensor and have each replica controlled by a different task. The tasks would then have to communicate in order to compare results. These results would inevitably be slightly different, and therefore some form of matching algorithm would be required.

Summary

This case study has been included to illustrate some of the issues discussed in this book. Unfortunately a single, relatively small, application cannot exercise all the important concepts that have been covered. In particular, issues of size and complexity are clearly not addressable within this context.

Nevertheless, it is hoped that the case study has helped to consolidate the reader's understanding of a number of topics, for example:

- top-down design and decomposition;
- concurrency and the models of interprocess communication;
- forward error recovery techniques and fault-tolerant design;
- periodic and sporadic processes;
- priority assignment and scheduling analysis;
- distributed programming.

Further reading

The case study given in this chapter is discussed by several other authors in the following references:

- Burns, A. and Lister, A. M. (1991) A framework for building dependable systems, *Computer Journal*, 34(2), 173–181.
- Joseph, M. (ed.) (1996) *Real-Time Systems: Specification, Verification and Analysis*. Englewood Cliffs, NJ: Prentice Hall (electronic version available from <http://www.cs.york.ac.uk/rts/RTS-SVABook.html>).
- Sloman, M. and Kramer, J. (1987) *Distributed Systems and Computer Networks*. Englewood Cliffs, NJ: Prentice Hall.

Exercises

- 15.1** If water is seeping into the mine at approximately the same rate as the pump is taking water out, the high water interrupt could be generated many times. Will this affect the behaviour of the software?
- 15.2** Identify which of the task interactions, given in the mine control system design, should be atomic actions. How could the solution be modified to support these actions?
- 15.3** Modify the solution given in this chapter so that it can be executed on a distributed system. Assume that each of the top-level objects, given in Figure 15.3, is an active partition.
- 15.4** To what extent can the solution to Exercise 15.3 be analysed for its timing properties?
- 15.5** Can the data logger determine the actual order of events that have occurred? If not, how could the code be modified to give a valid global ordering? What are the implications for a distributed implementation?
- 15.6** In the analysis of the mine control system, what would be the consequences of running the clock at 100 ms? (or 10 ms?)
- 15.7** Undertake a sensitivity analysis on the mine control tasks set. Taking each task in turn consider by how much its computation time must increase before the task set becomes unschedulable. Express this value as a percentage of the original C value.
- 15.8** If the deadlines for the CO sensor and air-flow sensor were both 50 then the system would not be schedulable. How could the fact that both sensors have the same period be exploited to obtain a schedulable system?

Chapter 16

Conclusions

The distinguishing characteristic of real-time systems is that correctness is not just a function of the logical results of program execution but of the time at which these results are produced. This one characteristic makes the study of real-time systems quite separate from other areas of computing. The importance of many real-time systems also places unique responsibilities on the practitioner. As more and more computers are being embedded in engineering applications, the greater is the risk of human, ecological or economic catastrophe. These risks arise from the problem of not being able to prove (or at least convincingly demonstrate) that all temporal and functional constraints will be met in all situations.

Real-time systems can be classified in a number of ways. First is the degree to which the application can tolerate tardiness in the system's responses. Those which have some flexibility are termed *soft* real-time systems; those with temporal rigidity are called *hard*. A deadline of three hours may be hard but easily attainable; one of three microseconds (hard or soft) presents the developer with considerable difficulty. Where deadlines, or response times, are very short the system is often called *real* real-time.

A non-real-time system can wait almost indefinitely for processors and other system resources. As long as such a system possesses *liveness* then it will execute appropriately. This is not the case with a real-time system. Because time is bounded and processors are not infinitely fast, a real-time program must be seen to be executing on a system with limited resources. It becomes necessary, therefore, to schedule the use of these resources between competing requests from different parts of the same program; a far from trivial endeavour.

Other characteristics of a typical modern real-time system are:

- they are often geographically distributed;
- they may contain a very large and complex software component;
- they must interact with concurrent real-world entities;
- they may contain processing elements which are subject to cost, power, size or weight constraints.

It follows from the very nature of most real-time applications that there is a stringent requirement for high reliability. This can also be formulated as a need for dependability and safety. Often, there is an almost symbiotic relationship between the computer system and its immediate environment. One cannot function without the other, as in a fly-by-wire aircraft. To give high levels of reliability requires fault-tolerant hardware and software.

There is a need for tolerance of loss of functionality and missed deadlines (even for hard real-time systems).

The combination of temporal requirements, limited resources, concurrent environmental entities and high reliability requirements (together with distributed processing) presents the system engineer with unique problems. Real-time and embedded systems engineering is now recognized as a distinct discipline. It has its own body of knowledge and theoretical foundation. From an understanding of the science of large real-time systems, the following are emerging:

- specification techniques that can capture temporal and fault tolerance requirements;
- design methods that have at their heart temporal requirements, and that can deal with methods of providing fault tolerance and distribution;
- programming languages and operating systems that can be used to implement these designs.

This book has been concerned with the characteristics and requirements of real-time systems, fault tolerance techniques, models of concurrency, time-related language features, resource control and low-level programming techniques. Throughout, attention has been given to the facilities provided by current real-time languages, in particular Ada, Java (and its Real-Time extensions) and C (augmented with the POSIX Real-Time and Threads extensions). Table 16.1 summarizes the facilities provided by these languages.

Ada is still the most appropriate language for high-integrity system and those systems which have hard real-time constraints. Real-Time Java now rivals Ada in terms of its expressive power. However, there are still some concerns over the efficiency of interpreting Java byte code. Indeed, it may well be the case that Real-Time Java, the language, is appropriate, but Real-Time Java, the virtual machine, is not. Consequently, either programs will need to be compiled rather than interpreted, or a hardware-assisted virtual machine will be needed. Perhaps where Real-Time Java does have the edge over Ada is with its support for more dynamic soft real-time systems. It is in this application domain that the language will find its initial use, particularly in those applications where portability is essential.

The C language, augmented with a real-time operating system (with or without POSIX compliancy), will continue to be used for those small embedded systems which are resource constrained.

Of course some systems will be written in more than one language. This is a result of the requirement to use legacy code, and the recognition that it is not appropriate to use the same language for all classes of applications. For example, a real-time application with a significant user interface component might be written in a mixture of Ada and Java. An 'intelligent' real-time system might require a rule-based component for which the most appropriate language might be Prolog.

To give an actual example of a real-time system, a case study has been described. Of necessity this was a relatively small system. Many of the challenges facing real-time computing are, however, manifest only in large complex applications. For example, consider the International Space Station. The primary function of its computer systems is mission and life support. Other activities include flight control (particularly of the orbital transfer vehicle), external monitoring, the control and coordination of experiments, and

	Ada	Real-Time Java	C/Real-Time POSIX
Support for programming in the large	Packages Generics Interfaces Tagged types	Packages Generics Interfaces Classes	Files
Support for concurrent programming	Tasks Rendezvous Protected types	Threads Synchronized methods and statements	Processes Threads Mutexes
Facilities for fault-tolerant programming	Exceptions ATC	Exceptions ATC Events	Signals
Real-time facilities	Clocks Timing events and delays	Clocks Timers and sleep	Clocks Timers and sleep
Scheduling facilities	Coherent priority model Dynamic priorities Group budgets EDF	Coherent priority model Dynamic priorities Processing group parameters	Coherent priority model Dynamic priorities Sporadic servers
Model of device handling	Shared memory	Limited shared memory	None defined
Profiles	Restricted tasking Ravenscar	Safety critical subsets	PSE profiles

Table 16.1 Summary of the facilities provided by Ada, Real-Time Java and C/Real-Time POSIX.

the management of the mission database. A particularly important aspect of the on-board software is the interface it presents to the flight personnel.

The on-board execution environment for the space station has the following pertinent characteristics.

- It is large and complex (that is, there are a large variety of activities to be computerized).
- It has non-stop execution.
- It has a long operational life (perhaps over 30 years).
- It will experience evolutionary software changes (without stopping).

- It must have highly dependable execution.
- It has components with hard and soft real-time deadlines.
- The distributed system contains heterogeneous processors.

To meet the challenges of this kind of application, the science of real-time systems must continue to develop. There are still many research themes to explore. Even the current state of understanding, which has been the focus of attention in this book, is rarely put into practice. In a search for the 'next generation' of real-time systems, Stankovic identified several research issues (Stankovic, 1988). Although much progress has been made since 1988, the following research topics are still crucial to the development of the discipline.

- Specification and verification techniques that can handle the needs of real-time systems with a large number of interacting components.
- Design methodologies that consider timing properties from the very beginning of the design process.
- Programming languages with explicit constructs to express time-related behaviour (in particular, relating to distributed computation).
- Scheduling algorithms that can handle complex process structures and resource constraints, timing requirements of varying granularity, and probabilistic guarantees.
- Real-time queuing models.
- Run-time support, or operating system functions, designed to deal with fault-tolerant resource usage.
- Tool support for predicting the worst-case and average execution times for software on complex modern processors.
- The integration of worst-case and average case performance metrics.
- Communication architectures and protocols for efficiently dealing with messages that require timely delivery across the Internet.
- Architecture support for fault tolerance and dynamic reconfiguration.
- Integrated support for Artificial Intelligence (for example, machine learning) components.
- Programming language and operating system support for atomic actions, recovery blocks, conversations or group communication protocols.
- Programming languages with explicit support for change management (i.e. the ability to do software upgrades to non-stop systems).
- Real-time virtual machines to support 'write once run anywhere' real-time applications with stringent timing constraints.
- Real-time reflective (self-modifying) architectures allowing applications to adapt their behaviour in response to changing environments.

It is to be hoped that some readers of this book will be able to contribute to these research themes.

References

- Adve, S. V. and Gharachorloo, K. (1996). Shared memory consistency models: A tutorial, *IEEE Computer* **29**(12): 66–76.
URL: citeseer.ist.psu.edu/adve95shared.html
- Allworth, S. and Zobel, R. (1987). *Introduction to Real-Time Software Design*, MacMillan.
- Ammann, P. and Knight, J. (1988). Data diversity: An approach to software fault tolerance, *IEEE Transactions on Computers* **C-37**(4): 418–425.
- Anderson, T. and Lee, P. (1990). *Fault Tolerance: Principles and Practice*, 2nd edn, Prentice Hall International.
- ARINC AEE Committee (1999). Avionics application software standard interface.
- Audsley, N. and Burns, A. (1998). On fixed priority scheduling, offsets and co-prime task periods, *Information Processing Letters* **67**: 65–69.
- Audsley, N., Burns, A., Richardson, M., Tindell, K. and Wellings, A. J. (1993a). Applying new scheduling theory to static priority pre-emptive scheduling, *Software Engineering Journal* **8**(5): 284–292.
- Audsley, N., Tindell, K. and Burns, A. (1993b). The end of the line for static cyclic scheduling?, *Proceedings of the 5th Euromicro Workshop on Real-Time Systems*, IEEE Computer Society Press, Oulu, Finland, 36–41.
- Avizienis, A. and Ball, D. (1987). On the achievement of a highly dependable and fault-tolerant air traffic control system, *Computer* **20**(2): 84–90.
- Avizienis, A., Lyu, M. and Schutz, W. (1988). Multi-version software development: A UCLA/Honeywell joint project for fault-tolerant flight control systems, *CSD-880034*, Department of Computer Science, University of California, Los Angeles.
- Avizienis, A., Laprie, J.-C., Randell, B. and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing, *IEEE Transactions on Dependable and Secure Computing* **1**(1): 11–33.
- Baker, T. P. (1991). Stack-based scheduling of realtime processes, *Real Time Systems* **3**(1): 67–99.

- Baruah, S. (2004). Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors, *IEEE Transactions on Computers* **53**(6): 781–784.
- Baruah, S. (2006). Resource sharing in EDF-scheduled systems: A closer look, *IEEE Real-Time Systems Symposium (RTSS)*, 379–387.
- Baruah, S., Mok, A. and Rosier, L. (1990a). Preemptive scheduling of hard real-time sporadic tasks on one processor, *IEEE Real-Time Systems Symposium (RTSS)*, 182–190.
- Baruah, S., Rosier, L. and Howell, R. (1990b). Algorithms and complexity concerning the preemptive scheduling of periodic tasks on one processor, *Journal of Real-Time Systems* **4**(2): 301–324.
- Baruah, S., Cohen, N., Plaxton, G. and Varvel, D. (1996). Proportionate progress: A notion of fairness in resource allocation, *Algorithmica* **15**(6): 600–625.
- Bate, I. and Burns, A. (1997). Timing analysis of fixed priority real-time systems with offsets, *9th Euromicro Workshop on Real-Time Systems*, 153–160.
- Bernat, G. and Burns, A. (1999). New results on fixed priority aperiodic servers, *IEEE Real-Time Systems Symposium (RTSS)*, 68–78.
- Berry, G. (1989). Real time programming: Special purpose or general purpose languages, in G. Ritter (ed.), *Proceedings of Information Processing '89*, Elsevier Science Publishers, 11–18.
- Bini, E., Buttazzo, G. and Buttazzo, G. (2007). Rate monotonic scheduling: The hyperbolic bound, *IEEE Transaction in Computer Systems* **52**(7): 933–942.
- Bloom, T. (1979). Evaluating synchronization mechanisms. *Proceedings of the Seventh Symposium on Operating Systems Principles*, ACM, 24–32.
- Boehm, H.-J. (2005). Threads cannot be implemented as a library, *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation*, ACM, New York, NY, USA, 261–268.
- Booch, G. (1986). *Software Engineering with Ada*, 2nd edn, The Benjamin/Cummings Publishing Company, Inc.
- Brilliant, S., Knight, J. and Leveson, N. (1987). The consistent comparison problem in N-version software, *ACM Software Engineering Notes* **12**(1): 29–34.
- Brilliant, S., Knight, J. and Leveson, N. (1990). Analysis of faults in an N-version software experiment, *IEEE Transactions on Software Engineering* **16**(2): 238–247.
- Brinch-Hansen, P. (1981). Edison – A multiprocessor language, *Software, Practice and Experience* **11**(4): 325–361.
- Buhr, P. A. (1995). Are safe concurrency libraries possible?, *Communications of the ACM* **38**(2): 117–120.
URL: citeseer.ist.psu.edu/buhr95are.html
- Bull, G. and Lewis, A. (1983). Real-Time BASIC, *Software, Practice and Experience* **13**(11): 1075–1092.
- Burns, A. and Lister, A. M. (1991). A framework for building dependable systems, *Computer Journal* **34**(2): 173–181.

- Burns, A. and Wellings, A. J. (1995). *Hard Real-Time HOOD: A Structured Design Method for Hard Real-Time Ada Systems*, Elsevier.
- Burns, A., Prasad, D., Bondavalli, A., Giandomenico, F. D., Ramamritham, K., Stankovic, J. and Stringini, L. (2000). The meaning and role of value in scheduling flexible real-time systems, *Journal of Systems Architecture*, **46**: 305–325.
- Campbell, R. and Randell, B. (1986). Error recovery in asynchronous systems, *IEEE Transactions on Software Engineering* **1**(8): 811–826.
- CCITT (1980). CCITT high level language (CHILL) recommendation Z.200.
- Chen, L. and Avizienis, A. (1978). N-version programming: A fault-tolerance approach to reliability of software operation, *Digest of Papers, The Eighth Annual International Conference on Fault-Tolerant Computing*, Toulouse, France, 3–9.
- Conway, M. E. (1963). Design of a separable transition-diagram compiler, *Communications of the ACM* **6**(7): 396–408.
- Coulouris, G., Dollimore, J. and Kindberg, T. (2005). *Distributed Systems, Concepts and Design*, 4th edn, Harlow, UK, Addison Wesley.
- Davis, R. and Wellings, A. J. (1995). Dual priority scheduling, *IEEE Real-Time Systems Symposium (RTSS)*, 100–109.
- Davis, R., Zabos, A. and Burns, A. (2008). Efficient exact schedulability tests for fixed priority pre-emptive systems, *IEEE Transactions on Computers* **57**(9): 1261–1276.
- de la Puente, J., Alonso, A. and Alvarez, A. (1996). Mapping HRT-HOOD designs to Ada 95 hierarchical libraries, *Ada-Europe'96 Conference*, Springer-Verlag, 78–88.
- Dijkstra, E. (1965). Solution of a problem in concurrent program control, *Communications of the ACM* **8**(9): 569.
- Dijkstra, E. (1968). Cooperating sequential processes, in F. Genuys (ed.), *Programming Languages*, Academic Press.
- Dijkstra, E. (1975). Guarded commands, nondeterminacy, and formal derivation of programs, *CACM* **18**(8): 453–457.
- dos Santos, O. M. and Wellings, A. (2008). Run time detection of blocking time violations in real-time systems, *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 347–356.
- Douglass, B. (1999). *Doing Hard Time: Developing Real-Time Systems with UML, Objects Frameworks and Patterns*, Addison-Wesley.
- Eckhardt, D., Caglayan, A., Knight, J., Lee, J., McAllister, D., Vouk, M. and Kelly, J. (1991). An experimental evaluation of software redundancy as a strategy for improving reliability, *IEEE Transactions on Software Engineering* **17**(7): 692–702.
- Esterel Technologies (2005). The Esterel v7 reference manual version v7 30 – initial IEEE standardization proposal. **URL:** <http://www.esterel-eda.com/style-EDA/files/papers/Esterel-Language-v7-Ref-Man.pdf>
- Evangelist, M., Francez, M. and Katz, S. (1989). Multiparty interactions for interprocess communication and synchronization, *IEEE Transactions on Software Engineering* **15**(11): 1417–26.
- Garman, J. (1981). The bug heard round the world, *Software Engineering Notes* **6**(3): 3–10.

- Goel, A. and Bastini, F. (1985). Software reliability, *IEEE Transactions on Software Engineering* **SE-11**(12): 1409–1410.
- Goossens, J., Funk, S. and Baruah, S. (2003). Priority-driven scheduling of periodic task systems on multiprocessors, *Real Time Systems: The International Journal of Time-Critical Computing* **25**(2–3): 187–205.
- Gray, J. (1986). Why do computers stop and what can be done about it?, *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, 3–12.
- Gregory, S. and Knight, J. (1985). A new linguistic approach to backward error recovery, *The Fifteenth Annual International Symposium on Fault-Tolerant Computing Digest of Papers*, 404–409.
- Halbwachs, N., Caspi, P., Raymond, P. and Pilaud, D. (1991). The synchronous dataflow programming language Lustre, *Proceedings of IEEE*, 1305–1320.
- Hatton, L. (1997). N-version design versus one good version, *IEEE Software* **14**(6): 71–76.
- Hecht, H. and Hecht, M. (1986). Fault-tolerant software, in D. Pradhan (ed.), *Fault-Tolerant Computing Theory and Techniques Volume II*, Prentice Hall, 659–685.
- Henzinger, T., Horowitz, B. and Kirsch, C. (2001). Giotto: A time-triggered language for embedded programming, *Proceedings of the International Workshop on Embedded Software, Lecture Notes in Computer Science* **2211**: 166–184.
- Hoare, C. (1974). Monitors – an operating system structuring concept, *CACM* **17**(10): 549–557.
- Hoare, C. (1978). Communicating sequential processes, *CACM* **21**(8): 666–677.
- Hoogeboom, B. and Halang, W. (1992). The concept of time in the specification of real-time systems, in K. Kavi (ed.), *Real-Time Systems: Abstractions, Languages and Design Methodologies*, IEEE Computer Society Press, 19–38.
- Horning, J. J., Lauer, H. C., Melliar-Smith, P. M. and Randell, B. (1974). A program structure for error detection and recovery, in E. Gelenbe and C. Kaiser (eds), *Lecture Notes in Computer Science 16*, Springer-Verlag, 171–187.
- IEEE (2004). Posix: the open group base specifications issue 6, *IEEE/1003.1*, IEEE.
- Joseph, M. (ed.) (1996). *Real-Time Systems: Specification, Verification and Analysis*, Prentice Hall.
- Joseph, M. and Pandya, P. (1986). Finding response times in a real-time system, *BCS Computer Journal* **29**(5): 390–395.
- Klein, M. H., Ralya, T. A., Pollak, B., Obenza, R. and Harbour, M. G. (1993). *A Practitioner's Handbook for Real-Time Analysis: A Guide to Rate Monotonic Analysis for Real-Time Systems*, Kluwer Academic Publishers.
- Kligerman, E. and Stoyenko, A. (1986). Real-Time Euclid: A language for reliable real-time systems, *IEEE Transactions on Software Engineering* **SE-12**(9): 941–949.
- Knight, J., Leveson, N. and St Jean, L. (1985). A large scale experiment in N-version programming, *Digest of Papers, The Fifteenth Annual International Symposium on Fault-Tolerant Computing*, Michigan, USA, 135–139.

- Kramer, J., Magee, J., Sloman, M. and Lister, A. (1983). CONIC: an integrated approach to distributed computer control systems, *IEE Proceedings (Part E)*, 1–10.
- Lamport, L. (1997). How to make a correct multiprocess program execute correctly on a multiprocessor, *IEEE Transactions on Computers* **46**(7): 779–782.
URL: citeseer.ist.psu.edu/article/lamport93how.html
- Laprie, J. (1995). Dependable – its attributes, impairments and means, in B. Randell, J. Laprie, H. Kopetz and B. Littlewood (eds), *Predictable Dependable Computing Systems*, Springer Verlag, 3–24.
- Lauer, H. and Needham, R. (1978). On the duality of operating system structure, *Proceedings of the Second International Symposium on Operating System Principles*, IRIA, 3–19.
- Lauer, H. and Satterwaite, E. (1979). The impact of Mesa on system design, *Proceedings of the 4th International Conference on Software Engineering*, IEEE, 174–182.
- le Guernic, P., Gautier, T., le Borgne, M. and le Maire, C. (1991). Programming real applications in Signal, *Proceedings of IEEE*, 1321–1313.
- Lee, I. and Gehlot, V. (1985). Language constructs for distributed real-time programming, *IEEE Real-Time Systems Symposium (RTSS)*, 57–56.
- Lee, P. (1983). Exception handling in C programs, *Software, Practice and Experience* **13**(5): 389–406.
- Lee, P., Ghani, N. and Heron, K. (1980). A recovery cache for the PDP-11, *IEEE Transactions on Computers* **C-29**(6): 546–549.
- Lehman, M. and Belady, L. (1985). The characteristics of large systems, *Program Evolution – Processes of Software Change*, APIC Studies in Data Processing No. 27, Academic Press, 289–329.
- Lehoczky, J. P., Sha, L. and Strosnider, J. K. (1987). Enhanced aperiodic responsiveness in a hard real-time environment, *IEEE Real-Time Systems Symposium (RTSS)*, 261–270.
- Leung, J. and Whitehead, J. (1982). On the complexity of fixed-priority scheduling of periodic, real-time tasks, *Performance Evaluation (Netherlands)* **2**(4): 237–250.
- Leveson, N. (1986). Software safety: Why, what and how, *ACM Computing Surveys* **18**(2): 125–163.
- Linux Manual Page (2006). sched_setaffinity().
URL: : http://www.die.net/doc/linux/man/man2/sched_setaffinity.2.html
- Littlewood, B. and Strigini, L. (1993). Validation of ultrahigh dependability for software-based systems, *Communications of the ACM* **36**(11): 69–80.
- Littlewood, B., Popov, P. and Strigini, L. (2001). Modelling software design diversity – a review, *ACM Computing Surveys* **33**(2): 177–208.
- Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment, *JACM* **20**(1): 46–61.
- Locke, C. (1992). Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives, *Real-Time Systems* **4**(1): 37–53.

- Lomet, D. (1977). Process structuring, synchronisation and recovery using atomic actions, *Proceedings ACM Conference Language Design for Reliable Software SIGPLAN*, 128–137.
- Lopez, J. M., Diaz, J. L. and Garcia, D. F. (2004). Utilization bounds for EDF scheduling on real-time multiprocessor systems, *Real-Time Systems Journal* **28**(1): 39–68.
- Martin, D. (1982). Dissimilar software in high integrity applications in flight controls, *AGARD Symposium on Software for Avionics*, 36:1–36:13.
- Matsuoka, S. and Yonezawa, A. (1993). Analysis of inheritance anomaly in object-oriented concurrent programming languages, in G. Agha, P. Wegner and A. Yonezawa (eds), *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 107–150.
- Meyer, B. (1992). *Eiffel: The Language*, Prentice Hall.
- Object Management Group (2002). CORBA component model, *OMG Document formal/2002-06-65*, Object Management Group.
- Oh, D.-I. and Baker, T. P. (1998). Utilization bounds for N-processor rate monotone scheduling with static processor assignment, *Real-Time Systems Journal* **15**: 183–192.
- Open Group/IEEE (2004). POSIX: The open group base specifications issue 6, ieee std 1003.1, 2004 edition, *IEEE/1003.1 2004 Edition*, The Open Group. URL: <http://www.opengroup.org/onlinepubs/009695399/>
- Parnas, D. (1994). Software aging, *Proceedings of the 16th International Conference on Software Engineering*, 279–287.
- Peterson, G. (1981). Myths about the mutual exclusion problem, *Information Processing Letters* **12**(3): 115–116.
- Pugh, W. (2000). The Java memory model is fatally flawed, *Concurrency: Practice and Experience* **12**(6): 445–455.
URL: citeseer.ist.psu.edu/pugh00java.html
- Purtilo, J. and Jalote, P. (1991). An environment for developing fault-tolerant software, *IEEE Transactions on Software Engineering* **17**(2): 153–159.
- Randell, B. (1975). System structure for software fault tolerance, *IEEE Transactions on Software Engineering* **SE-1**(2): 220–232.
- Randell, B. (1982). *The Origins of Digital Computers: Selected Papers*, Springer Verlag.
- Randell, B., Lee, P. and Treleaven, P. (1978). Reliability issues in computing system design, *ACM Computing Surveys* **10**(2): 123–165.
- Randell, B., Laprie, J.-C., Kopetz, H. and Littlewood, B. (eds) (1995). *Predictably Dependable Computing Systems*, Springer Verlag.
- Regehr, J. (2007). Safe and structured use of interrupts in real-time and embedded software, in I. Lee, J. Y.-T. Leung and S. H. Son (eds), *Handbook of Real-Time and Embedded Systems*, Chapman and Hall/CRC, 16-1–16-12.
- Rogers, P. and Wellings, A. J. (2000). An incremental recovery cache supporting software fault tolerance mechanisms, *Computer Systems Science and Engineering* **15**(1): 33–48.

- Rubira-Calsavara, C. and Stroud, R. (1994). Forward and backward error recovery in C++, *Object-Oriented Systems* **1**(1): 61–85.
- Sha, L., Rajkumar, R. and Lehoczky, J. P. (1990). Priority inheritance protocols: An approach to real-time synchronisation, *IEEE Transactions on Computers* **39**(9): 1175–1185.
- Shrivastava, S. (1978). Sequential Pascal with recovery blocks, *Software, Practice and Experience* **8**(2): 177–186.
- Shrivastava, S., Mancini, L. and Randell, B. (1987). On the duality of fault tolerant structures, *Lecture Notes in Computer Science*, Vol. 309, Springer Verlag, 19–37.
- Sloman, M. and Kramer, J. (1987). *Distributed Systems and Computer Networks*, Prentice Hall.
- Stankovic, J. (1988). Misconceptions about real-time computing: A serious problem for next generation systems, *IEEE Computer* **21**(10): 10–19.
- Tindell, K., Burns, A. and Wellings, A. J. (1994). An extendible approach for analysing fixed priority hard real-time tasks, *Real-Time Systems* **6**(2): 133–151.
- Venners, B. (1999). *Inside the Java 1.2 Virtual Machine*, Osborne McGraw-Hill.
- Walker, W. and Cragon, H. (1995). Interrupt processing in concurrent processors, *Computer* **28**(6): 36–46.
- Wellings, A. J. and Burns, A. (1997). Implementing atomic actions in Ada 95, *IEEE Transactions on Software Engineering* **23**(2): 107–123.
- Werum, W. and Windauer, H. (1985). *Introduction to PEARL Process and Experiment Automation Realtime Language*, Friedr. Vieweg Sohn.
- Whitaker, W. (1978). The U.S. Department of Defense common high order language effort, *ACM SIGPLAN Notices* **13**(2): 19–29.
- Xerox Corporation (1985). Mesa language manual version 5.0.
- X/Open Company Ltd (1996). *X/Open Guide: Architecture Neutral Distribution Format*, X/Open Company Ltd, UK.
- Young, S. (1982). *Real Time Languages: Design and Development*, Ellis Horwood.
- Yuh-Jzer, J. and Smolka, S. A. (1996). A comprehensive study of the complexity of multiparty interaction, *Journal of the ACM* **43**(1): 75–115.
- Zhang, F. and Burns, A. (2008). Schedulability analysis for real-time systems with EDF scheduling, *Technical Report YCS 426*, University of York.

Index

- abort deferred operations 257, 258
- abort deferred regions 296
- aborting a task 110, 245–6, 258, 295–6
- absolute delays 318–20
- abstract model of device handling 502, 503–4
- accept statement 198
- acceptance tests 47, 49
- access permissions 44
- active control agents 285
- active faults 29
- active objects 102, 539
- active priority 428, 429
- Ada 23
 - 'Caller 109
 - 'Count 167
 - 'Storage Pool 522
 - 'Terminated 110, 197
- abort deferred operations 257, 258
- abort deferred regions 296
- abort statement 258
- accept statement 198
- analogue-to-digital converters 510–13
- anonymous tasks 109
- aperiodic activities 341–2
- asynchronous notification 255–66
- asynchronous transfer of control (ATC) 255–7, 324
- at clause 505
- atomic actions 235–6, 258–66, 294
- Attach_Handler 508
- Baker's algorithm support 446–51
- Barrier 163
- categorization pragmas 123
- clocks 311–12, 313, 468–70, 486–7
- concurrent programming 105–11
- Constraint_Error 63, 64, 70, 76
- controlled types 75
- deadlines 444–6, 459–62
- deferrable server 379, 399
- delay 318–19
- device registers 504–7
- dispatching policy 446
- distributed objects 212
- dynamic priorities 434–6
- else alternative 203
- entries 196–7
- entry families 197
- exception handling 63, 65, 69–78, 199–200, 257
- execution-time servers 379, 399
- group budgets 479–85
- heap management 522–3
- I/O device access 513–14
- I/O jitter 349–52
- interfaces 169–70, 200–1
- interrupt handling 507–10
- last wishes 74–5
- message passing 196–205, 213–14
- mine control case study 546–64
- Modify requests 288–9
- multiprocessor systems 122–3, 213–14

Ada (*Continued*)

normal library package 213

package

Ada.System 427, 506

Ada.Calendar 311

Ada.Dispatching.EDF 444

Ada.Dynamic_Priorities 435

Ada.Exceptions 71

Ada.Execution_Time 468

Ada.Execution_Time.Group_Budgets
480

Ada.Execution_Time.Timers 470

Ada.Interrupts 509

Ada.Interrupts.Names 510

Ada.Real_Time 313

Ada.Real_Time.Timing_Events
345

Ada.Task_Identification 108

Ada.Task_Termination 110

Calendar 311

Exceptions 69

Remote_Call_Interface 213

Standard 63

System 427, 506

System.Machine_Code 514

System.RPC 215

System.Storage_Pools 523

parameter passing 78

partition identification 214

periodic activities 338–9

pragmas

Atomic 182

Atomic.Components 182

Interrupt_Policy 428

Locking_Policy 428

Preelaborate 123

Priority 427

Pure 123

Remote_Call_Interface 213

Remote.Types 123

Restrictions pragma 430–4

Shared.Passive 123

Volatile 182

Volatile.Components 182

priority 427

priority assignment 398

priority-based scheduling 427

Program_Error 63

protected entry 197

protected interfaces 169

protected objects 163–70

Ravenscar profile 430–4

Real_Time 313

Real_Time Systems Annex 427

Record representation clause
405–6

remote procedure calls 213–14, 215

rendezvous 199–200

representation clauses 504–7

request types 288–9

requeue facility 296–302

resource control 291–4, 302–3

restricted tasking 431–3

scheduling

earliest deadline first 444–52

fixed-priority 427–30

mixed 453–4

select statement 202–5

select then abort 445–6

semaphores 150–2, 168–9

server programming 478–85

shared variables 182

sporadic event overruns 471–3

Storage_Error 70

storage pools 522, 523

synchronized interfaces 169

task 105

activation 108

rendezvous 197

termination 110

types 106

task entry 196

Task_Id 109

Tasking_Error 108, 197

task abortion 258

task control 144

task identifiers 108

task identification 109–10

task interfaces 169

task synchronization and

communication 504–14

terminate alternative 110, 203

time-triggered events 344–5,
460–2

- timeouts 321
- timing checks 42
- timing events 345
- unchecked conversion 513
- unhandled exception 110
- watchdog timer 460–2
- when others 72–3, 77, 78
- air traffic control 35, 38
- Airbus 35
- allocation 121
- alternative modules 47
- Amdahl's law 96
- American Strategic Defense Initiative (SDI) 34
- analogue-to-digital (ADC) converters 510–13
- ANDF 21
- aperiodic activities 4
 - in Ada 341–2
 - in C/Real-Time Posix 342
 - in Real-Time Java 342–4
- aperiodic servers 476
- aperiodic temporal scopes 329
- application error detection 42
- Application Program Interfaces (APIs) 14
- arbitrary failure 31
- architectural design 16, 534, 535–6, 538–45
- Arianne 5 disaster 31
- arithmetic overflow 59
- assembly languages 22
- assertions 43
- associated overheads 49–50
- asymmetric naming 195, 302–3
- async-cancel unsafe 253
- async-signal unsafe 253
- asynchronous exceptions 62, 243
- asynchronous message passing 194–5
- asynchronous notification 245–78, 503
 - in Ada 255–66
 - in C/Real-Time Posix 247–55
 - need for 246–7
 - and operating systems 245–6
 - in Real-Time Java 266–78
 - resumption model 245, 247
 - termination model 245–6
- asynchronous task control 435
- asynchronous transfer of control (ATC) 245
 - in Ada 255–7, 324
 - in C/Real-Time Posix 253–4
 - in Real-Time Java 268–74
- Asynchronously Interruptible (AI) methods 269–74
- atomic actions 43–4, 228–45
 - in Ada 235–6, 258–66, 294
 - backward error recovery 241–2, 259–62
 - boundary definition 231
 - in C/Real-Time Posix 232–5, 254–5
 - concurrency 231
 - conversations 241–2
 - double interactions 294
 - and exceptions 243–5
 - failure atomicity 230
 - forward error recovery 242–3, 263–6
 - internal atomic actions 244–5
 - in Java 237–40
 - Real-Time Java 275–8
 - nested 229, 231, 244
 - properties 228–9
 - recoverable 230, 240–5
 - requirements 231
 - resource allocation 230, 232
 - resource control 286
 - server tasks 229–30
 - synchronization atomicity 230
 - two-phase atomic actions 229–30, 232
- atomic transactions 230–1
- atomicity 138
- Augustine, Saint 308
- avoidance synchronization 287, 296
- avoidance-based primitives 288, 295
- backward error recovery 44–6, 227, 241–2, 245, 259–62
- bandwidth preserving schemes 380
- barriers 163, 166
- base priority 428
- best-effort scheduling 406
- bin packing problem 367
- binary semaphores 150

- blocked interrupts 507
- blocking 382–6, 404–5
- Bloom’s criteria 287
- Bohrbugs 30
- boundary definition 231
- bounded buffers 157–8, 162, 164–5
- bounded errors 434
- bounded liveliness 327
- budgets 41
- buffers 139, 157–8, 162, 164–5, 501–2
- bugs 29–30
- busy periods 376
- busy waiting 139–42, 148–9

- C 22, 61, 84–5, 517–19
 - exception handling 60
 - exceptions 84
 - low-level programming 517
- C++ 22, 63
 - exceptions 63
- C/Real-Time Posix 1
 - aperiodic activities 342
 - asynchronous notification 247–55
 - asynchronous transfer of control 253–4
 - atomic actions 232–5, 254–5
 - barriers 163
 - blocking a signal 248
 - clockid_t type 317
 - clocks 316, 320, 467–8
 - concurrent programming 116–20
 - condition variables 157–8, 160–3
 - deadline miss detection 454–5
 - error conditions 60–1
 - exception handling 60–1
 - fixed-priority scheduling (FPS) 436–8, 439
 - generating a signal 251
 - handling a signal 250
 - I/O jitter 352–4
 - ICPP 428
 - ignoring a signal 251
 - itimerspec struct 346
 - message passing 206–10
 - mq_attr type 207
 - mqd_t type 207
 - multiprocessor systems 124–5
 - mutexes 160–3
 - periodic activities 338–9
 - pid_t type 438
 - priority 436
 - priority inheritance 437
 - processor affinity 124–5
 - profiles 436
 - pthread 116–20
 - pthread cancellation 254
 - pthread detaching 119
 - pthread_attr_t type 116, 117
 - pthread_cancel method 116
 - pthread_cond_t type 160
 - pthread_condattr_t type 160
 - pthread_create method 116
 - pthread_exit method 118
 - pthread_join method 118
 - pthread_kill method 253
 - pthread_mutex_t type 160
 - pthread_mutexattr_t type 160
 - pthread_t type 118
 - pthreads 116, 253
 - read/write locks 163
 - sched_param struct 438
 - scheduling 437
 - sem_t type 153
 - semaphores 152–5
 - sigaction method 252
 - sigaction struct 249
 - sigevent struct 248
 - siginfo_t type 249
 - signals 247–55
 - pthreads 253
 - sigqueue 253
 - sigset_t type 249
 - sigval union 248
 - sporadic server 437
 - server programming 476–7
 - thread cancellation 253–4
 - time_t type 316
 - timer_t type 346
 - timers 345
 - timespec struct 346
 - time-triggered events 345–6
 - timeouts 320, 321
 - watchdog timer 464
 - vfork 103

- Calendar 311
- catch all 66
- catch statement 81
- causal ordering 308
- ceiling protocols 386–90
- channel-program control 498
- checkpointing 45
- child task 102
- CHILL 23, 65, 66
- circular wait 304
- class construct 17
- classes of exceptions 62
- client/server model 195
- client stub 210, 211, 216
- clock drift 309
- clocks
 - in Ada 311–12, 313
 - in C/Real-Time Posix 316, 320
 - in distributed systems 309–10
 - environment time frame 309
 - execution-time 467–71
 - internal hardware 309
 - interrupt handling 415–16
 - monotonic 312, 316–17
 - real-time 316–17, 417–19
 - in Real-Time Java 312–15
 - and scheduling 415–16, 417–19
 - skew 309
 - see also time*; timing faults
- coarse parallelism 100
- cobegin 104–5
- coding 16, 43, 534
- cohesion 18
- commission failure 32
- Communicating Sequential Processes (CSP) 327
- comparison points 37, 38
- comparison status indicators 37, 38
- comparison vectors 37, 38
- competing tasks 99, 228, 285
- complex systems 13–14
- compositional decomposition 17
- concurrency 231
- concurrent control 10–11
- Concurrent Pascal 157
- concurrent programming 95–133, 228
 - in Ada 105–11
 - in C/Real-Time Posix 116–20
 - constructs 99
 - in Java 111–16
 - language supported 131
 - multiprocessor systems 121–5, 179–82
 - Non-Uniform Memory Architectures 97
 - operating systems supported 96, 97, 128, 131
 - processes 96–9
 - reasons for 95–6
 - Run-Time Support System 97, 98, 147, 148
- concurrently raised exceptions 243–4
- condition synchronization 138–9, 146, 287
- condition variables 157–8, 160–3
- conditional critical regions 156–7
- conditional entry calls 323
- conditional wait 288
- configuration 121
- consistent comparison problem 39
- constraint errors 31
- constructs 99
- context switching 414–16, 498–500
- continuous change 14
- control
 - active and passive agents 285
 - concurrent control 10–11
 - feed-forward controller 12
 - feedback controller 12
 - interrupt-driven 497–501
 - priority control 501
 - real-time control facilities 9–10
 - status-driven 497
 - see also* asynchronous transfer of control (ATC); resource control
- Control Area Network (CAN) 412
- control and status registers 501–2
- conversations 241–2
- cooperating tasks 99, 228
- cooperative dispatching 369
- cooperative scheduling 393–4
- Coral 66 22

- CORBA (Common Object Request Broker Architecture) 213, 217–19
- CORBA Component Model (CCM) 219
- counting semaphores 146
- coupling 18
- critical instant 370, 372, 395–6
- critical section 138, 141
- cumulative drift 319
- cyber-physical systems 9
- cycle stealing 498
- cyclic executives 366–7, 425–6
- cyclic objects 539

- daemon threads 115
- damage confinement and assessment 41, 43–4, 457
- data buffer registers 501–2
- data logging 8
- data race condition 143
- data retrieval 8
- DatagramSocket class 124
- Deadline monotonic 380
- deadlines 444–6
 - case study 537–8
 - inversion 404
 - missed deadlines 454–5, 458, 459–67, 488–9
 - scheduling 329, 380–2, 391–2
- deadlock 149, 155, 304, 389–90
- decomposition 17, 43–4
- deferrable server 379, 399
- Deferrable Server (DS) scheduling 379–80, 399–400
- deferred preemption 369
- delay 317
 - absolute 318
 - relative 317
- delay alternative 203
- delaying a task 317–20
- delivery of an interrupt 507
- dense time 308
- dependability 54–5
- deserter problem 241
- design 15–19
 - detailed 16, 534
 - diversity 37, 50
 - errors 570–1
 - HRT-HOOD method 534–9
 - independence 40–1
 - logical architecture 535–6, 538–44
 - physical architecture 536, 545–6
- destroy method 296, 348
- development cycle 15–19
- devices
 - drivers 519–21
 - handling 502, 503–4
 - identification 500
 - registers 495–6, 504–7, 514–16, 517–19
 - see also* hardware
- differential equations 15
- digital control 8
- Direct Memory Access (DMA) 497–8
- disable method 347
- discrete time 309
- dispatching policy 446
- distributed objects 212–13
- distributed systems *see* multiprocessor systems
- diversity of design 37, 50
- DMA 415, 497–8
- DMPO (deadline monotonic priority ordering) 380–2
- domain 63–5
- domino effect 46, 227, 241
- dormant faults 29
- double interactions 294
- DPS 258–9, 465–5
- driver code 361
- driver process 37
- dual-priority scheduling 380
- Dynamic Deferrable Server 405
- dynamic priorities 434–6
- dynamic reasonableness checks 43
- dynamic redundancy 36, 41–6, 49, 50–2
 - and timing faults 457–9
- dynamic scheduling 366, 405–7, 434–6
- Dynamic Sporadic Server 405
- dynamic tasks 100

- earliest deadline first (EDF) scheduling
 - 368, 400–5, 406, 443–52
- Edison 157
- efficiency 22
- Eiffel 67
- Einstein, Albert 308
- else alternative 203
- embedded systems 1, 3
- encapsulation facilities 502, 503
- entities 102
- entries 196–7
- entry calls 166–9, 323
- entry families 197
- environment time frame 309
- environmental error detection 42
- equipment interfaces 6–7, 11–12
- errors
 - constraint errors 31
 - in design 570–1
 - detection 36, 41, 42–3, 50
 - deadline misses 457, 459–67
 - and exception handling 60–2
 - recovery 41, 44–6, 50, 227, 241–6, 259–66
 - timing faults 458, 485–92
 - at requirements stage 34
 - see also* faults
- Esterel 360–1
- event handlers 245, 266–8, 344–9
- event-based reconfiguration 490–2
- event-triggered systems 3–4
- exception tree 243–4
- exceptions and exception handling 50–2, 59–89, 466–7
 - in Ada 63, 65, 69–78, 199–200, 257
 - asynchronous 62, 243
 - atomic actions 243–5
 - in C++ 63
 - in C 84–5
 - in C/Real-Time Posix 60–1
 - classes 62
 - concurrently raised 243–4
 - domain 63–5
 - error procedures 61–2
 - failure exceptions 51
 - forced branch 61
 - hybrid model 66, 68
 - interface exceptions 51
 - in Java 63, 78–83
 - modern approaches 62–9
 - non-local goto 61–2
 - in older languages 60–2
 - and operating systems 68–9
 - propagation 65–6, 73–4, 78, 82
 - recovery blocks 85–8
 - requirements 59
 - resumption (notify) model
 - 66–8, 243
 - skip return 61
 - at statement level 65
 - status flags 60–1
 - symbol lookups 51
 - synchronous 62
 - termination (escape) model
 - 66, 68, 243
 - thread-related 115–16
 - unhandled 66
 - unusual return values 60–1
- execution time servers 399–400, 405, 476
- execution-time clocks 467–71, 486–7
- explicit Kutta method 48–9
- expressive power 21, 287–8
- extended rendezvous 194, 197
- external state 29
- fail controlled 32
- fail late 32
- fail never 32
- fail safe 34, 329, 458
- fail silent 32
- fail soft 34
- fail stop 32
- fail uncontrolled 32
- failure
 - atomicity 230
 - and exceptions 51
 - fail soft 34
 - fail-safe systems 34, 35
 - of hardware 571–2
 - modes 31–2
- fault, error, failure chain 29
- fault model 29

faults

- avoidance 33
- damage confinement and assessment 41, 43–4, 457
- fault, error, failure chain 458–9
- fixed-priority scheduling (FPS) 394–5
- granularity 38–9, 100
- ideal system components 51
- prevention 33–4
- recovery blocks 46–50, 85–8
- removal 33
- tolerance 33, 34–5, 458–9, 570–2
- treatment 41, 46, 458
- types 29
- see also* timing faults
- Federal Aviation Administration 35
- feed-forward controller 12
- feedback controller 12
- fibers 97
- fine grain parallelism 100
- firewalling 43
- firm deadlines 3
- firm real-time systems 329
- fixed-priority scheduling (FPS) *see* scheduling
- flags 139, 140–1
- flat tasks 100
- flexibility of languages 21
- floating-point arithmetic 13
- forced branch 61
- fork and join 103–4
- FORTRAN 22
- forward error recovery 44, 46, 50, 242–3, 245, 263–6
- FPS 370
- functional requirements 536
- Gantt charts 372
- garbage collection 522
- Giotto 361–3
- global dispatching 121
- global multiprocessor scheduling 409–11
- Global Positioning Systems (GPS) 309
- goto 61–2
- graceful degradation 34–5
- granularity 38–9, 100

group budgets 479–85

group servers 476, 478

growth models 53

guarded blocks 63

guarded commands 202

guardian/dependent tasks 101

Hamming Codes 44

happens-before relation 180–1

hard real-time systems 2–3, 329

hardware

equipment interfaces 6–7, 11–12

failure 571–2

I/O mechanisms 495–502, 513–14

see also devices

heap management 521–6

heap objects 524

Heisenbugs 30

hierarchy schedulers 476

hierarchy of tasks 101

high-level concurrent languages 22–3

high-level language primitive 500

hold and wait 304

HRT-HOOD design method 534–9

hybrid model of exceptions 66, 68

hybrid scheduling 406

ICPP 388

Ada 428

C/Real-Time POSIX 437

Real-Time Java 441

I/O device access 495–502, 513–14

I/O jitter 3, 329, 349–56

ideal synchronous hypothesis 360

ideal system components 51

immediate ceiling priority protocol 386, 388–9, 428

immortal memory 523–4

immortal objects 525

implicit Kutta method 48–9

imprecise computations 246

impromptu failure 32

incremental checkpointing 45

indefinite postponement 150, 304

independence of design 40–1

independent tasks 99

indivisible actions 228, 229, 231

- inexact voting 39
- inheritance anomaly 178–9
- input jitter 3, 329, 349–56
- input/output mechanisms 495–502, 513–14
- inter-task communication 137, 228
- inter-task relationships 101
- interactive systems 329
- Interface Definition Language (IDL) 218
- interfaces 51, 169–70, 272–3
- intermittent faults 29
- internal atomic actions 244–5
- internal hardware clocks 309
- interrupt control 501
- interrupt handling 415–16, 507–10, 516–17
- interrupt identification 500
- interrupt masks 501
- interrupt representation 503–4
- interrupt vector table 500
- interrupt-driven control 497–501
- Interrupted Exception 269
- Interruptible 273
- Interruptible interface 272–3
- intertask communication 99
- isEnabled method 270
- isInterrupted method 268
- Java 21, 23
 - atomic actions 237–40
 - catch statement 81–2
 - class synchronization 172
 - concurrency utilities
 - concurrent.locks 176
 - concurrency utils
 - ReentrantLock class 177
 - concurrent programming 111–16
 - currentThread 115
 - destroy method 296, 348
 - distributed objects 213
 - exception handling 63, 78–83
 - finally clause 82
 - inheritance anomaly 178–9
 - interrupt method 270
 - InterruptedException 268
 - isInterrupted method 268
- last wishes 82
- Lock interface 176
- locks 176–8
- memory model 180–1
- message passing 214–17
- monitors 171
- multiprocessor systems 123–4, 214–17
- Naming class 216–17
- networking 123–4
- notify method 173
- notify All method 173
- Object class 173
- readers-writers problem 174–6
- real-time threads 336–8
- registry 216
- release parameters 337
- remote objects 124, 214
- RemoteServer class 216
- Resume method 142
- Runnable interface 111
- semaphores 152
- skeleton 216
- static modifier 115
- synchronized blocks 171
- synchronization 171–9
- this 172
- Thread class 111–15, 142–3
- Throwable class 63
- timeouts 321
- try-block 63–4, 81–2
- UnicastRemoteObject class 216
- waiting and notifying 172–4
- see also* Real-Time Java
- java.lang
 - thread 111
- java.net 123
- java.rmi 226
- job 370
- Jovial 22
- kernel-level threads 97
- Kutta method 48–9
- large systems 13–14
- last wishes 74–5, 82
- latency of interrupts 507

- level interrupt control 501
- library-level threads 97
- life cycle models 534
- linear time 308
- Linux 103
- livelock 140, 304
- liveness 149–50, 304
- local drift 319
- lockout 150, 304
- locks 176–8
- logical architecture 535–6, 538–44
- Logical Execution Time (LET) model 362
- longjmp 61, 84
- low-level programming 11–12
- Lustre language 360

- mailbox 195
- major cycles 366
- manufacturing systems 5–6
- Mars Pathfinder 384
- marshalling parameters 211
- mask interrupt control 501
- masking redundancy 36, 41, 49
- measuring reliability 52–3
- memory areas 523–6
- memory leak 30
- memory management 521–6
- memory model 180–1
- memory parameters 338
- memory-mapped I/O 496
- Mesa 23
- message passing 137–8, 193–221
 - accept statement 198
 - in Ada 196–201, 202–5
 - asynchronous 194–5
 - in C/Real-Time Posix 206–10
 - distributed object model 212–13
 - distributed systems 210–19
 - in Java 214–17
 - message queues 206–10
 - message structure 196
 - non-determinism 205–6
 - process synchronization 193–5
 - remote call procedures 210–12
 - remote invocation 194, 212
 - rendezvous 199–200
 - select statement 202–5
 - selective waiting 201–2, 205–6
 - synchronization primitives 205–6, 287–8, 295, 365
 - synchronous 194, 321
 - task interfaces 200–1
 - task naming 195–6, 302–3
 - timeouts 321–4
- message queues 206–10
- message structure 196
- message-based condition synchronization 504
- middleware 211
- minimum inter-arrival time (MIT) 341, 473–4
- minor cycles 366
- mishaps 53
- mixed scheduling 453–4
- mobile systems 8, 12
- mode changes 10, 246, 490–2
- model checking 326–7
- model driven architectures (MDA) 16
- Modula-1 157
- Modula-2 66, 68
- modular decomposition 43–4
- modularity facilities 502, 503
- monitors 157–9, 286, 295
- monotonic clocks 312, 316–17
- multi-media systems 7–8
- multiparity interactions 228
- multiprocessor systems 122–5, 214–17
 - clocks 309–10
 - concurrent programming 121–5, 179–82
 - message passing 210–19, 213–14
 - mutual exclusion 412–13
 - scheduling 408–13
- Mururao nuclear test 34
- mutexes 160–3, 286
- mutual exclusion 138–9, 140
 - and ceiling protocols 389–90
 - and multiprocessor systems 412–13
 - and semaphores 146, 155

- N Modular Redundancy (NMR) 36
- N*-version programming 36–41, 49–51
 - compared to recovery blocks 49–50

- costs 41
- independence of design 40–1
- initial specification 40
- vote comparison 39
- necessity tests 368
- nested atomic actions 229, 231, 244
- nested monitor calls 159
- nested tasks 100
- network scheduling 411–12
- non-determinism 205–6
- non-deterministic construct 202
- non-deterministic scheduling 206
- non-functional requirements 537
- non-local goto 61–2
- non-preemptive scheduling 369
- non-preemptible operations 148
- Non-Uniform Memory Architectures (NUMA) 97
- notify model 66
- nuclear reactors 31, 54
- numerical computation support 12–13
- object-oriented programming 169–70
- objects
 - abstraction 17–18
 - active 102, 539
 - cyclic 539
 - distributed 212–13
 - immortal 525
 - passive 538
 - protected 163–70, 539
 - reactive 102
 - remote 124, 212–13, 214
 - scoped 525
 - sporadic 539
 - suspended 144
 - and tasks 102–3
- offsets 395–7
- online scheduling 405–7
- operating systems
 - asynchronous notification 245–6
 - concurrent programming 96, 97, 128, 131
 - exceptions and exception handling 68–9
 - ORB (Object Request Broker) 217–18
 - original ceiling priority protocol 386–7
 - output jitter 3, 329, 349–56
 - overhead scheduling factors 414–19
 - parallelism 95, 97, 100, 228
 - parameter marshalling 211
 - parameter passing 78
 - parent/child tasks 101
 - partition identification 214
 - partitioned multiprocessor scheduling 409–11
 - partitioning 121, 122
 - passive control agents 285
 - passive entities 102
 - passive objects 538
 - Pathfinder 384
 - Patriot Missile Software Problem 30
 - PEARL 23, 358
 - pending interrupts 507
 - period displacement 520
 - periodic activities 3
 - in Ada 338–9
 - in C/Real-Time Posix 338–9
 - in Real-Time Java 339–41
 - PeriodicParameters class 340
 - Periodic Server (PS) scheduling 399–400
 - periodic temporal scopes 329
 - permanent faults 29
 - physical architecture 536, 545–6
 - Platonism 308–9
 - polling device identification 500
 - port-mapped I/O 496
 - portability of languages 21
 - Portable Object Adapter (POA) 219
 - Posix *see* C/Real-Time Posix
 - power-aware systems 413–14
 - preemptive scheduling 369, 405, 426
 - primary modules 47
 - priority assignment 397–8
 - priority ceiling protocols 386–90
 - priority control 501
 - priority inheritance 384
 - priority inversion 382–4, 404
 - priority-based scheduling *see* scheduling
 - procedure calls 503

process control 4–5
 process synchronization 193–5
 processing groups 338, 478
 ProcessingGroupParameters 479
 processor affinity 121
 processor demand criteria (PDC) 401–3
 processor status word (PSW) 502
 producer-consumer systems 139
 program counter (PC) 502
 propagation 65–6, 73–4, 78, 82
 protected interfaces 169–70
 protected objects 163–70, 539
 protected procedures 508–10
 protected resources 102, 285, 286–7
 protection mechanisms 44
 protective redundancy 36
 pthread 116–20
 pthread_sigmask 253
 pure library package 123

 QPA (Quick Processor demand Analysis)
 403–4
 quality improvements 33
 quantity semaphores 150

 race condition 143
 radio time signals 309
 raising an exception 50
 rate monotonic priority assignment 370
 Ravenscar profile 430–4
 reactive objects 102
 reactive systems 3
 read/write locks 163, 166
 readability of languages 20–1
 readers-writers problem 174–6
 ready queue 427
 Real-Time Basic 68
 real-time clocks 316–17, 417–19
 real-time control facilities 9–10
 Real-Time Euclid 320–1, 356–7, 490–2
 real-time events 344–9
 Real-Time Java 1–2
 aperiodic activities 342–4
 asynchronous notification 266–78
 asynchronous transfer of control
 268–74

Asynchronously Interruptible (AI)
 methods 269–74
 atomic actions 275–8
 clocks 312–15, 470–1
 deadline misses 462–4, 488–9
 device registers 514–16
 event handling 266–8
 fixed-priority scheduling (FPS)
 438–43
 heap management 523–6
 I/O jitter 354–6
 interrupt handling 516–17
 Interruptible interface 272–3
 memory areas 523–6
 mine control case study 564–70
 minimum inter-arrival time (MIT)
 473–4
 periodic activities 339–41
 processing groups 478
 real-time threads 116
 server programming 478
 sleep method 319
 time-triggered events 347–9
 timeouts 325–6
 waitForNextPeriod (wFNP)
 339–41, 463
 see also Java
 real-time systems 1–25
 characteristics 9–15
 in communication, command and
 control 6
 cyber-physical 9
 definition 2–4
 development cycle 15–19
 equipment interfaces 6–7, 11–12
 languages 20–3
 in manufacturing 5–6
 mobile 8, 12
 multi-media 7–8
 in process control 4–5
 record keeping 7
 response times 2–3, 9
 see also response-time analysis
 real-time tasks 335, 336–8, 344
 real-time threads 116
 Real-Time UML 33
 reasonableness checks 43

- reconfiguration 121
- record keeping 7
- recoverable atomic actions 230, 240–5
- recovery blocks 46–50, 85–8
- recovery cache 45
- recovery lines 46
- recovery points 45, 46–7
- Reductionism 308–9
- redundancy *see* dynamic redundancy;
static redundancy
- registers 216, 501–2
- relative delays 317–18
- relaxed memory models 180
- release jitter 390–1
- release parameters 336
- reliability 14–15
 - dynamic redundancy 36, 41–6, 49,
50–2, 457–30–2, 451–9
 - failure modes 31–2
 - growth models 53
 - measuring 52–3
 - N-version programming 36–41,
49–51
 - see also* errors; faults
 - remote invocation 194, 212
 - remote objects 124, 212–13, 214
 - remote procedure calls 210–12,
213–14, 215
 - Remote_Types 123
 - rendezvous 194, 197, 199–200
 - replication checks 42
 - representation clauses 504–7
 - representation of tasks 103–5
 - request order 289
 - request parameters 290–4
 - request priority 295
 - request types 288–9
 - requeue facility 294, 296–302
 - requirements specifications 16–17, 534
 - reservation-based systems 476
 - reserved interrupts 508
 - resource allocation 230, 232, 285, 286–7
 - resource control 285–305, 291–4, 302–3
 - asymmetric naming 302–3
 - and atomic actions 286
 - expressive power 287–8
 - request order 289
 - request parameters 290–4
 - request priority 295
 - request types 288–9
 - requeue facility 294, 296–302
 - server state 289–90
 - resource management 230, 232, 285,
286–7
 - resource servers 229–30
 - resource usage 303, 474–5
 - response time analysis (RTA) 2–3, 9,
374–8, 386, 401
 - resumption model 66–8, 243, 245, 247
 - reversal checks 42–3
 - robot arm 106
 - Ada 106
 - C/Real-Time POSIX 119
 - Java 111
 - Round-Robin scheduling 436, 437
 - RTL/2 22
 - Run-Time Support System (RTSS) 97,
98, 147, 148
 - Runnable 111
 - safety 14–15, 53–4
 - see also* reliability
 - scheduling 206, 246–7, 365–420, 545–6
 - in Ada 427–30
 - aperiodic tasks 378–80, 405
 - bandwidth preservation 380
 - blocking 382–6, 404–5
 - busy periods 376
 - in C/Real-Time Posix 436–8, 439
 - clocks 415–16, 417–19
 - cooperative 393–4
 - cyclic executives 366–7, 425–6
 - device drivers 519–21
 - dual-priority 380
 - dynamic 366, 405–7, 434–6
 - earliest deadline first 368, 400–5,
406, 443–52
 - execution time servers 399–400
 - fault tolerance 394–5
 - fixed-priority 368, 370–1
 - insufficient priorities 398–9
 - interference 374–5
 - mixed schemes 453–4
 - multiprocessor 408–13

scheduling (*Continued*)

- non-deterministic 206
 - non-preemptive 369
 - offsets 395–7
 - online 405–7
 - overhead factors 414–19
 - parameters 338
 - power-aware systems 413–14
 - preemptive 369, 405, 426
 - priority assignment 370, 397–8
 - priority ceiling protocols 386–90
 - priority inheritance 384
 - priority inversion 382–4, 404
 - real-time clock handler 417–19
 - in Real-Time Java 438–43
 - release jitter 390–1
 - response time analysis 374–8, 386, 401
 - Round-Robin 436, 437
 - server methods 379–80, 399–400, 405
 - sporadic tasks 378–80, 416–17
 - static 366
 - task interactions 382–6
 - task values 406
 - task-based 367–70
 - tests 368, 371–4, 400–1, 403–4
 - time-lines 372
 - transient overload 379
 - value-based 368
 - worst-case execution time 407–8, 416
 - see also* deadlines
- scoped memory 524
- scoped objects 525
- secure languages 20
- selective waiting 201–2, 205–6
- semaphores 145–55
- in Ada 150–2, 168–9
 - binary 150
 - in C/Real-Time Posix 152–5
 - condition synchronization 146
 - counting semaphores 146
 - criticisms 155
 - implementation 148–9
 - in Java 152
 - liveness provision 149–50
 - mutual exclusion 146, 155
 - quantity semaphores 150
 - suspended tasks 147–8
 - test and set instructions 148
- sequential systems implementation
- languages 22
- server object 102, 103
- server programming 478–85
- server state resource control 289–90
- server stub 211, 216
- servers 285, 379, 476, 478
- atomic actions 229–30
 - execution time servers 399–400, 405, 476
 - scheduling methods 379–80, 399–400, 405
- setjmp 61, 84
- shared variables 182
- SIGABRT 247
- SIGALARM 247
- SIGALRM 517
- Signal language 360
- signals
- blocking 248–50
 - in C/Real-Time Posix 247–55
 - generating 251
 - handling 250–1
 - ignoring 251
 - and threads 252–3
- SIGRTMAX 247
- SIGRTMIN 247
- simplicity of languages 21
- simulation 19
- skeleton 216
- skip return 61
- sleep method 319
- soft real-time systems 2–3, 329
- software aging 30
- software crisis 22–3
- socket class 123
- sockets 210
- Sojourner 384
- Space Shuttle 36
- Special Relativity 308
- spin locks 139
- spinning 139, 412–13
- sporadic activities 4

- sporadic event overruns 471–4, 488
- Sporadic Servers (SS) 379–80, 399–400, 437, 476–7
- sporadic tasks 378–80, 416–17, 503
- sporadic temporal scopes 329
- sporadic objects 539
- SR language 293
- stack management 526
- Stack Resource Policy (SRP) 405
- standard time 310
- starvation 150, 304
- state restoration 45
- static redundancy 36, 41, 49
- static scheduling 366
- static tasks 100
- status device identification 500
- status flags 60–1
- status interrupt control 501
- status-driven control 497
- stiff equations 48
- storage pools 522, 523
- structural checks 43
- structure of tasks 100
- sufficiency tests 368
- suspend and resume 142–5
- suspended tasks 147–8
- suspension objects 144
- symbol lookups 51
- synchronization
 - atomicity 230
 - avoidance synchronization 287, 296
 - busy waiting 139–42, 148–9
 - condition synchronization 138–9, 146, 287
 - inheritance anomaly 178–9
 - in Java 171–9
 - primitives 205–6, 287–8, 295, 365
 - process synchronization 193–5
 - resource management 286–7
 - time protocols 310
 - see also* message passing
- synchronized interfaces 169–70, 286
- synchronized resources 285
- synchronous exceptions 62
- synchronous languages 360
- synchronous message passing
 - 194, 321
- synchronous task control 144, 435
- system overheads 414
- system repair 46
- task-based scheduling 367–70
- tasks 96–9
 - aborting 110, 245–6, 258, 295–6
 - abstraction 18
 - cobegin 104–5
 - communication 137, 228, 504–14
 - competing 99, 228, 285
 - context switches 414–16, 498–500
 - cooperating 99, 228
 - delaying 317–20
 - execution 97–8, 99–103
 - explicit declaration 105
 - fork and join 103–4
 - hierarchies 101
 - identification 100, 108–10, 115, 302
 - initialization 100
 - interactions 99, 382–6
 - naming 195–6, 302–3
 - and objects 102–3
 - real-time 335, 336–8, 344
 - relationships 101
 - representation 103–5
 - structure 100
 - suspended 147–8
 - synchronization 504–14
 - termination 100–2, 110, 115
 - values 406
- temporal scopes 328–32, 335
 - in DPS 258–9
 - in Esterel 360–1
 - in Giotto 361–3
 - in Pearl 358
 - in Real-Time Euclid 356–7
- termination model 66, 68, 243, 245–6
- termination of tasks 100–2, 110, 115
- test and set instructions 148
- testing 16, 19, 33–4, 534
 - acceptance test 47, 49
 - scheduling 368, 371–4, 400–1, 403–4
- thread cancellation 253–4
- thread-related exceptions 115–16
- throwing an exception 80–1

- time
 - checks 42
 - cumulative drift 319
 - delaying a task 317–20
 - local drift 319
 - notion of time 307–9
 - radio time signals 309
 - specifying requirements 326–7
 - standards 310
 - synchronisation protocols 310
 - watchdog timer 42, 460–2, 464
 - see also* clocks; timing faults
- Time Division Multiple Access (TDMA) 412
- time-aware systems 3
- time-lines 372
- time-triggered events 3
 - in Ada 344–5, 460–2
 - in C/Real-Time Posix 345
 - in Real-Time Java 347
- Timed Automation 327
- timed entry calls 323
- timers
 - C/Real-Time POSIX 345
 - Real-Time Java 347
- timeouts 320–6
 - and actions 324–6
 - in Ada 321
 - in C/Real-Time Posix 320, 321
 - in Java 321
 - and message passing 321–4
 - in Real-Time Java 325–6
 - shared variable communication 320–1
- timing checks 42
- timing failure 31, 32
- timing faults 457–93
 - damage confinement 475–85
 - deadline misses 459–67, 488–9
 - dynamic redundancy 457–9
 - error detection 465–7
 - error recovery 485–92
 - event-based reconfiguration 490–2
 - mode changes 490–2
 - resource usage overruns 474–5
 - sporadic event overruns 471–4, 488
 - worst-case execution overrun 467–71, 485–7
- transactions 44
- transitive blocking 386
- transient faults 29
- transient overload 379
- transparent execution 121
- Triple Modular Redundancy (TMR) 36
- try-block 63–4, 81–2
- two-phase atomic actions 229–30, 232
- two-stage suspend operation 143–4
- unhandled exceptions 66
- Universal Coordinated Time (UTC) 309, 310
- Universal time 310
- unplanned events 53
- unusual return values 60–1
- usability of languages 21
- user interrupts 247
- user threads 115
- utilization-based scheduling tests 371–4, 400–1
- value failure 31
- value-based scheduling (VBS) 368
- variety 14
- vectored device identification 500
- volatile data 182
- vote comparison 39
- waitForNextPeriod (wFNP) 339–41, 463
- watchdog timer 42, 460–2, 464
- when others 72–3, 77, 78
- worst-case execution time 407–8, 416, 467–71, 485–7
- write locks 163, 166